Vince

ACCENT QNIX MANUAL

VOLUME ONE

Preliminary Draft for Review

June 18, 1985

Reviewers:

Nick Felisiak, Spider
Ellen Colwell
Bob Mitchell
Vicki Preston
Circulating copy
(Joyce Swaney, Don Scelza)

Instructions to Reviewers

The enclosed copy is yours to keep. Please make a copy of any pages on which you have comments and give the copies to Vince Guinto. Comments received by July 1 will be considered for Release 1; comments received after that date will be considered for the next release.

Note

The released document will be on 7" x 9" paper, copied by an outside firm. It will be packaged in the new style of binder labeled "Accent Qnix Manual" and will have printed index dividers.

Vince

cc for comments:
Don
Nick
Ellen
Vicki
Bob M.

PREFACE

QNIX is an emulation of the Bell Laboratories UNIX System V operating system, ~~QNIX was~~ developed by PERQ Systems Corporation ~~and Spider Systems, Edinburgh, Scotland.~~ It runs under Accent, an operating system for the PERQ workstation that was developed jointly by PERQ Systems Corporation and the Spice Project in the Computer Science Department at Carnegie-Mellon University. "Spice" is an acronym for Scientific Personal Integrated Computing Environment.

Don said to take this out, as we don't usually give sub-contractors credit. I made a copy for Nick w/o it. - Joyce

The documents in this manual are adapted from documents from Bell Laboratories.

Following Bell Laboratories' practice, the commands, routines, and facilities available under QNIX are documented in categories:

1 - user commands

2 - system calls

3C and 3S - subroutines and libraries

3X - specialized library routines

4 - file formats

5 - miscellaneous facilities

Items in these sections are referred to by the item name followed by the section number in parentheses; for example, as(1) refers to the as command in Section 1.

In addition to these six documents, this manual also contains a document that explains how QNIX differs from the UNIX System V operating system; a basic introductory document; documents on the text editor, the shell, the utilities Make, Augmented Make, Awk, LEX, and YACC; documents on the C language and the C library; a glossary; and a permuted index.

ACCENT QNIX MANUAL

VOLUME ONE

User Guide
1
Basics for Beginners
Text Editor Tutorial
Introduction to the Shell
Make
Augmented Make
Awk
Lex
YACC
Glossary
Index

VOLUME TWO

2
3C and 3S
3X
4
5
C Language
C Libraries
Software Reports

Vince

ACCENT QNIX MANUAL

VOLUME ONE

Preliminary Draft for Review

June 18, 1985

Reviewers:

Nick Felisiak, Spider
Ellen Colwell
Bob Mitchell
Vicki Preston
Circulating copy
(Joyce Swaney, Don Scelza)

Instructions to Reviewers

The enclosed copy is yours to keep. Please make a copy of any pages on which you have comments and give the copies to Vince Guinto. Comments received by July 1 will be considered for Release 1; comments received after that date will be considered for the next release.

Note

The released document will be on 7" x 9" paper, copied by an outside firm. It will be packaged in the new style of binder labeled "Accent Qnix Manual" and will have printed index dividers.

Vince

cc for comments:
Don
Nick
Ellen
Vicki
Bob M.

PREFACE

QNIX is an emulation of the Bell Laboratories UNIX System V operating system, ~~QNIX was~~ developed by PERQ Systems Corporation ~~and Spider Systems, Edinburgh, Scotland.~~ It runs under Accent, an operating system for the PERQ workstation that was developed jointly by PERQ Systems Corporation and the Spice Project in the Computer Science Department at Carnegie-Mellon University. "Spice" is an acronym for Scientific Personal Integrated Computing Environment.

Don said to take this out, as we don't usually give sub-contractors credit. I made a copy for Nick w/o it. - Joyce

The documents in this manual are adapted from documents from Bell Laboratories.

Following Bell Laboratories' practice, the commands, routines, and facilities available under QNIX are documented in categories:

1 - user commands

2 - system calls

3C and 3S - subroutines and libraries

3X - specialized library routines

4 - file formats

5 - miscellaneous facilities

Items in these sections are referred to by the item name followed by the section number in parentheses; for example, as(1) refers to the as command in Section 1.

In addition to these six documents, this manual also contains a document that explains how QNIX differs from the UNIX System V operating system; a basic introductory document; documents on the text editor, the shell, the utilities Make, Augmented Make, Awk, LEX, and YACC; documents on the C language and the C library; a glossary; and a permuted index.

If you have had experience using UNIX software, we suggest that you read the first document in this manual, "QNIX User Guide," which explains the differences between UNIX System V and QNIX. Then use the rest of the manual as a refresher and reference. If you have not used UNIX software, we recommend that you first read the documents "Basics for Beginners" and "Introduction to the Shell." Then read the rest of the manual as needed for the tasks you are doing.

If you are not familiar with the Accent operating system, it would be helpful to read the Accent manuals, especially the Accent User's Manual. Other manuals for Accent are the Accent Programming Manual, Accent Languages Manual, Accent Microprogramming Manual, and Accent System Administration Manual.

Throughout the documentation the term "PERQ" refers to all models of the PERQ workstation unless stated otherwise. When a distinction is made between the PERQ workstation and the PERQ2 workstation, the term "PERQ2" refers to both Model LN-3000 and LN-3500.

ACCENT QNIX MANUAL

VOLUME ONE

User Guide
1
Basics for Beginners
Text Editor Tutorial
Introduction to the Shell
Make
Augmented Make
Awk
Lex
YACC
Glossary
Index

VOLUME TWO

2
3C and 3S
3X
4
5
C Language
C Libraries
Software Reports

QNIX User Guide

Nick Felisiak

Spider Systems Ltd.

1 INTRODUCTION

QNIX is an emulation of the AT&T UNIX System V operating system that runs under Accent. It combines many of the standard UNIX facilities with the advantages of Accent.

This document describes the QNIX system and its similarities and differences with UNIX. The document is intended for users who are familiar with both Accent and UNIX.

2 THEORY

QNIX combines the Accent environment with the UNIX operating system. This combination allows the use of standard UNIX facilities and existing UNIX programs under the Accent environment.

In order to make QNIX compatible with Accent, it has been necessary to remove some of the features of UNIX. Most of the missing features involve system control and administration, and will not affect the normal user. Some UNIX functions (for example, the mail system) are provided by Accent, and are not duplicated in QNIX.

QNIX is a server process operating under Accent. The QNIX server process provides many of the functions of the UNIX kernel. It also supplies a set of library routines available to programs running in the QNIX environment. The server process and library routines provide a faithful emulation of UNIX System V, except as noted in the QNIX documentation. The UNIX C library is also supported under QNIX.

The implementation of existing UNIX programs should be straightforward with QNIX.

3 ESTABLISHING QNIX

To use the QNIX environment, a QNIX server process must be started. This server registers itself with the Accent Name Server, and will support QNIX processes initiated on the same machine. In a future release, it will be possible for a QNIX server to support QNIX processes on many machines running under different users.

3.1 Parameter File

When started, the QNIX server reads the file 'boot:qnixparams'. This file contains information about the QNIX environment and its relationship to Accent.

3.1.1 Well-known prefixes

Many file prefixes in UNIX are considered to be 'well-known'. For example, the directory '/tmp' is referenced by many programs, and UNIX programs are often searched for in the directory '/bin'.

Because of the importance of these well-known prefixes, QNIX provides a mechanism for defining them in the parameter file. Any file referenced from a QNIX program whose name starts with a well-known prefix will be searched for, not in the full Accent name space, but instead in a restricted QNIX name space which is part of the Accent filestore.

The first line in the QNIX parameter file is the full pathname for these well-known prefixes. For example, if a user decides to establish a 'Unix' partition to hold the well-known directories, the first line of the parameter file would be:

```
/sys/unix
```

Note that the prefix must be a full pathname, and may not contain a logical name.

Prefixes which are to be 'well-known' for QNIX are identified in the parameter file by lines containing the prefix names preceeded by a 'w'. For example,
the following lines define the prefixes '/tmp', '/bin', '/dev', '/usr', and '/lib':

```
/sys/unix
w /tmp
w /bin
w /dev
w /usr
w /lib
```

With these lines in the parameter file, a reference to '/bin/ls' from a QNIX program would be interpreted as accessing '/sys/unix/bin/ls'.

Prefixes can be more than one directory deep. If the only directory under '/usr' that was to be used by QNIX programs was '/usr/man', then '/usr' could be replaced by '/usr/man' in the above example.

3.1.2 Devices

The notion of devices is not fully supported in QNIX. However, two devices that are frequently referenced, the terminal ('/dev/tty') and the null device ('/dev/null') are supported by QNIX as major devices 0 and 1, respectively. They must be defined in the parameter file as follows:

```
d /dev/tty 0 0 d /dev/null 1 0
```

The inclusion of minor device numbers (the second number after each definition) is to allow for future development; minor devices are not currently implemented. Unlike UNIX, there are no block devices in QNIX, so there are no character device indicators in the parameter entry.

Note that '/dev' should be declared as a well-known prefix for consistency.

3.1.3 Named pipes

QNIX supports named pipes by allowing them to be defined in the parameter file. Pipes may also be entered in the filestore by 'mknod', but pipes created in this way are removed at the end of the QNIX session.

Pipes are identified in the parameter file by lines starting with 'f'. The lines are of the form:

```
f /sys/accent/systempipe
f /lib/mypipe
```

Note that the second pipe will appear in the directory '/sys/unix/lib', since '/lib' is defined as a well-known prefix in an earlier example.

Objects such as pipes and devices will appear in directory listings obtained by QNIX programs (such as 'ls'), but will NOT be visible to Accent programs (such as 'direct').

3.2 Starting a QNIX Server

A QNIX server can be started by the user or by a launch command. In either case, the server requires a window for the output of messages. These messages note the opening and closing of files and any error conditions that arise in the QNIX environment. The window may usually be left offscreen. However, it may be useful to display the window if a QNIX program is behaving unexpectedly.

Once the server is running, QNIX programs may connect to it. The server should not be stopped while QNIX programs are still connected; doing so will cause the programs to fail, and files held open by these programs will not be written back to disk.

4. THE QNIX FILE SYSTEM

QNIX files are identical in most respects to Accent files. They share the same names, except for the case of well-known prefixes, and they have the same format and permission control. There are, however, some differences.

The main difference between QNIX files and Accent files is that QNIX files can be shared between different QNIX processes simultaneously. The rules for sharing QNIX files are identical to UNIX file sharing rules, and are as follows:

> Files may have many readers and/or writers. Shared writers can interfere with each other catastrophically if uncoordinated.
>
> File connections survive across fork and exec system calls. A fork will give rise to a pointer shared by two processes.
>
> Shared pointers (whether in the same or different processes) behave within the system as a single object.
>
> Reads and writes are atomic. That is, a single call will give rise to a contiguous sequence of bytes in the file.

It is important to note that Accent programs may access files open to QNIX according to Accent's rules; the whole file will be read or written in a single transfer when opened or closed. This may lead to inconsistencies between Accent and QNIX. QNIX files are read in when first opened, and written back (if modified) when the last writer closes the file.

4.1 Directories, Inodes and I-Numbers

Since QNIX uses the Accent file system, it does not provide UNIX-style directories or inodes. However, these are simulated by QNIX.

Directories follow the standard format of a sixteen-bit i-number followed by a filename. To avoid ambiguity, filenames are extended to 80 characters, and the 'DIRSIZ' value in 'dir.h' has been amended accordingly.

The 'stat' (and 'fstat') system calls are supported, using information constructed from the Accent file system's data. However, some of the fields of these calls have slightly modified meanings. See the description of stat(2) in the QNIX Programmer Reference Manual for details.

Because there are no real inodes, there are also no real i-numbers. QNIX allocates i-numbers to files when they are opened, and a file will keep the same i-number while it is held open. If the file is closed and re-opened, the new connection will usually appear to have a new i-number. One consequence of this is that programs that read the i-number of files ('ls -i', for example) will give different values for files that have no continuous connection between successive runs. This property of QNIX is not considered a serious drawback.

4.2 Search Lists

The Accent file system includes the concept of searchlists--lists of directories in which to search for files opened for read access. QNIX preserves this facility, and allows an Accent pathname of the form

<searchlist>:<directory/directory/name>

When read access to a file is requested, the entire searchlist is examined, one element at a time, until a file matching the specified name is encountered. If no searchlist is specified, the 'default:' list is used.

When writing, only the top-most directory on the specified searchlist is considered. Again, if no list is specified, the top-most element of the 'default:' searchlist is used. The 'default:' list is always defined as 'current:'.

The 'chdir' system call updates the value of 'current:', and the shell 'pwd' command displays this variable.

5. ENVIRONMENT INTERFACE

QNIX uses the Accent Environment Manager to support its environment. Exported shell variables are stored as local string variables in the Environment Manager. The 'getenv' routine uses the environment manager.

Only string variables are visible through the QNIX interfaces. To use the Environment Manager fully, to use searchlists, or to declare global string variables, the Environment Manager interface must be used directly.

UNIX programs that access the 'environ' variable to handle their full environment may continue to do so under QNIX by calling the QNIX 'makeenv' routine, which calls the Environment Manager directly and constructs the required environment.

Accent environment lookups are not case sensitive ('var', 'Var', and 'VAR' are considered identical). The QNIX 'getenv' routine is also case insensitive. However, programs that use environment names internally will continue to be case sensitive. This difference will be most notable among programs in the UNIX shell.

6. SIGNALS, MESSAGES, EXCEPTIONS

QNIX extends the UNIX facilities for handling signals and provides a uniform way of handling interrupt messages for programs.

The number of available process signals in QNIX has been increased to 32. A number of additional signals are used by the Accent Process Manager. The default action for QNIX signals not used by UNIX is for the receiving program to ignore them.

The 'setpause' library routine provides an additional signal mechanism. Setpause is used to define which signals will interrupt system calls that are waiting for an event. Setpause allows the new signals to be used without affecting the operation of a UNIX program. By default, all signals will cause 'pause', 'wait', and some 'open' system calls to unblock.

QNIX provides an interface ('setuserhandler') to allow programs to handle interrupt messages. This interface MUST be used. Programs should not attempt to set interrupt handlers themselves; doing so will interfere with QNIX process and file control. Setuserhandler is similar to 'signal'--it allows a function to be defined that is invoked when an interrupt is to be serviced. The function is called with the message as its only parameter.

7 DEVELOPING C PROGRAMS FOR QNIX

Program development in QNIX combines the best of both UNIX and Accent. Specific points are discussed in this chapter.

It is the goal of the QNIX system to allow UNIX programs to run unchanged under QNIX, provided that they do not use privileged system calls or have excessive knowledge of the inner workings of UNIX. The QNIX Programmer Reference Manual should be consulted for details of the QNIX system interfaces.

7.1 Library Organization

Libraries under QNIX are built and used differently than under standard UNIX. In QNIX, a library is shared by all programs linked with it. To achieve this, QNIX uses a hierarchy of libraries. Each library may refer to those inferior to it in the hierarchy, and is in turn referred to by those superior to it. No library may have unsatisfied references when built.

Libraries are allocated fixed addresses in the store image of the C program. Because of this, if a library is rebuilt for any reason, all libraries superior to it must also be rebuilt. Rebuilding the lowest level libraries is therefore a potentially expensive operation.

Names are resolved according to the same hierarchy. If a name is already known is redefined in a superior library, the new definition is visible in all higher levels of the hierarchy. However, existing uses of the name will continue to reference the old definition of the name. The loader will issue a warning about any such redefinitions.

Libraries are built using the loader. The 'ar' and 'ranlib' utilities are not available in QNIX.

7.2 Compilation System

The QNIX compilation system is largely the same as that of UNIX. Extensions have been made to exploit Accent facilities where useful.

Searchlists 'lib:' and 'include:' are used in preference to the built-in pathnames for library files and include files, respectively. If these are undefined, the defaults used are the same as for standard UNIX systems: '/lib' and '/usr/lib' for libraries, and '/usr/include' for include files. If the defaults are to be used, these names should be defined as well-known prefixes.

7.3 Interaction with Accent Facilities

Usually, the QNIX facilities form an extension to those offered by Accent. There are some areas, however, where caution is needed when using Accent facilities from a QNIX program.

7.3.1 Memory management

The UNIX memory management facilities are supported in QNIX with some minor modifications.

The 'brk' and 'sbrk' system calls allocate store from a 4Mb region that is allocated the first time the calls are used. This store area is not necessarily coincidental with the address space of the compiled program; usually it will be completely distinct. This region is not expandable beyond 4Mb.

The 'malloc' subroutine does not use 'brk' to gain store; instead, it calls the Accent 'ValidateMemory' routine. Therefore, it is very unlikely that 'malloc' will fail under any circumstances.

The Accent memory management calls may be used directly, but care should be taken not to quote specific addresses to 'ValidateMemory', or to invalidate memory obtained by QNIX system calls.

NAME

intro – introduction to commands and application programs

DESCRIPTION

This section describes publicly-accessible commands. The commands are in alphabetical order.

COMMAND SYNTAX

Unless otherwise noted, commands described in this section accept options and other arguments according to the following syntax:

name [*option*(*s*)] [*cmdarg*(*s*)]

where:

name	The name of an executable file.
option	– *noargletter*(*s*) or, – *argletter*<>*optarg* where <> is optional white space.
noargletter	A single letter representing an option without an argument.
argletter	A single letter representing an option requiring an argument.
optarg	Argument (character string) satisfying preceding *argletter*.
cmdarg	Path name (or other command argument) *not* beginning with –, or – by itself indicating the standard input.

SEE ALSO

exit(2), wait(2).
The document "Basics for Beginners" in this manual.

DIAGNOSTICS

Upon termination, each command returns two bytes of status, one supplied by the system and giving the cause for termination, and (in the case of "normal" termination) one supplied by the program (see *wait*(2) and *exit*(2)). The former byte is 0 for normal termination; the latter is customarily 0 for successful execution and non-zero to indicate troubles such as erroneous parameters, bad or inaccessible data, or other problems. It is known as "exit code," "exit status," or "return code," and is described only where special conventions are involved.

BUGS

A number of commands do not adhere to the syntax described above.

WARNINGS

Some commands produce unexpected results when processing files containing null characters. These commands often treat text input lines as strings and therefore become confused upon encountering a null character (the string terminator) within a line.

NAME

as – PERQ C Code assembler

SYNOPSIS

as [**–o** objfile] [**–O**] [**–D**] [**–d**] file-name

DESCRIPTION

The *as* command assembles the named file. The following flags may be specified in any order:

–o *objfile* Put the output of the assembly in *objfile*. By default, the output filename is **a.out**.

–O Invoke the optimizer. Optimization enables the use of fast registers and generates smaller code.

–D Do not generate assembler debugging information (this is the default).

–d Generate assembler debugging information. The information is written to the standard output.

SEE ALSO

ld(1).

"PERQ C Programming" and "C System Interfaces" in the *Accent Languages Manual*, provided with the C compiler.

NAME

awk – pattern scanning and processing language

SYNOPSIS

awk [**–Fc**] [prog] [parameters] [files]

DESCRIPTION

Awk scans each input *file* for lines that match any of a set of patterns specified in *prog*. An action can be specified for each pattern in *prog* that will be performed if a match with that pattern occurs. The set of patterns can be given literally as *prog*, or can be in a file specified as **–f** *file*. The *prog* string should be enclosed in single quotes (') to protect it from the shell.

Parameters may be passed to *awk* in the form x=... y=... etc.

Files are read in the order specified. If no files are specified, the standard input is read. The filename – means the standard input. Each line of the files is matched against the pattern portion of every pattern-action statement in *prog*. Each time a line matches a pattern, the associated action is performed.

A line from an input file is made up of fields separated by white space. (Note: the field separator can be changed using FS; see below). The fields are denoted **$1**, **$2**, ...; **$0** refers to the entire line.

A pattern-action statement is of the form:

pattern { action }

If no action is given, the line is printed. If no pattern is given, the action is always performed.

An action is a sequence of statements. A statement can be one of the following:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
break
continue
{ [ statement ] ... }
variable = expression
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next
exit
```

Statements are terminated by semicolons, newlines, or right braces. An empty expression-list means the whole line. Expressions have string or numeric values as appropriate, and are built using the operators +, −, *, /, %, and concatenation (indicated by a blank). The C operators ++, −−, +=, −=, *=, /=, and %= are also available in expressions. Variables may be scalars, array elements (denoted x[i]), or fields. Variables are initialized to the null string. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. String constants are quoted (").

The *next* statement skips the remaining patterns on the current line of input. The *exit* statement skips the rest of the input.

The *print* statement prints its arguments on the standard output (or on a file if *>expr* is present), separated by the current output field separator, and terminated by the output record separator. The *printf* statement formats its expression list according to its format (see *printf*(3S)).

The built-in function *length* returns the length of its argument taken as a string, or of the whole line if no argument is specified. Other built-in functions are *exp*, *log*, *sqrt*, and *int*. The last truncates its argument to an integer. The function *substr*(*s*, *m*, *n*) returns the *n*-character substring of *s* that begins at position *m*. The function *sprintf*(*fmt*, *expr*, *expr*, ...) formats the expressions according to the *printf*(3S) format given by *fmt* and returns the resulting string.

Patterns are arbitrary boolean combinations (using !, ||, &&, and parentheses) of regular and relational expressions. Regular expressions must be surrounded by slashes and are of the form accepted by *egrep* (see *grep*(1)). Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions.

A pattern may consist of two patterns separated by a comma. In this case, the associated action is performed for all lines that fall between an occurrence of the first pattern and an occurrence of the second.

A relational expression is one of the following:

expression matchop regular-expression
expression relop expression

Relop is any of the six relational operators in C. Matchop is either ~ (for *contains*) or !~ (for *does not contain*).

A conditional is an arithmetic expression, a relational expression, or a boolean combination of these.

The special patterns BEGIN and END may be used to capture control before the first input line is read and after the last line is read. BEGIN must be the first pattern, END the last.

A single character *c* may be used to separate the fields of input lines by starting the program with:

```
BEGIN { FS = c }
```

or by using the **−F***c* option.

Variable names with special meanings include:

NF	The number of fields in the current record.
NR	The ordinal number of the current record.
FILENAME	The name of the current input file.
OFS	The output field separator (default blank).
ORS	The output record separator (default newline).
OFMT	The output format for numbers (default **%.6g**).

EXAMPLES

Print lines longer than 72 characters:

```
length > 72
```

Print first two fields of all lines in opposite order:

```
{ print $2, $1 }
```

Add up first column, print sum and average:

```
        { s += $1 }
END     { print "sum is", s, " average is", s/NR }
```

Print fields in reverse order:

```
{ for (i = NF; i > 0; --i) print $i }
```

Print all lines between start/stop pairs:

```
/start/, /stop/
```

Print all lines whose first field is different from previous one:

```
$1 != prev { print; prev = $1 }
```

Print file, filling in page numbers starting at 5:

```
/Page/ { $2 = n++; }
       { print }
```

command line: awk −f program n=5 input

SEE ALSO

grep(1), lex(1), sed(1).
malloc(3X).
The document "The awk Programming Language" in this manual.

BUGS

Input white space is not preserved on output if fields are involved.

There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate the null string ("") to it.

NAME

basename, dirname – deliver portions of pathnames

SYNOPSIS

basename string [suffix]
dirname string

DESCRIPTION

Basename deletes any prefix ending in / or : and the specified *suffix* (if present in *string*) from *string*, and prints the result on the standard output. It is normally used inside substitution marks (` `) within shell procedures. A version number suffix is always removed from *string* before other processing.

Dirname delivers all but the last level of the pathname in *string*.

EXAMPLES

The following example, invoked with the argument **/usr/src/cmd/cat.c**, compiles the named file and moves the output to a file named **cat** in the current directory:

```
cc $1
mv a.out `basename $1 '\.c'`
```

The following example sets the shell variable **NAME** to **/usr/src/cmd**:

```
NAME=`dirname /usr/src/cmd/cat.c`
```

SEE ALSO

sh(1).

BUGS

The *basename* of / is null and is considered an error.

NAME

cat – concatenate and print files

SYNOPSIS

cat [–u] [–s] [–v [–t] [–e]] file ...

DESCRIPTION

Cat reads each *file* in sequence and writes it to the standard output. For example:

cat file

prints *file*, and:

cat file1 file2 >file3

concatenates the first two files and writes the result to the third. If no input file is given, or if – is given, *cat* reads from the standard input file.

Output is buffered unless the **–u** option is specified. The **–s** option makes *cat* silent about non-existent files.

The **–v** option causes non-printing characters (with the exception of tabs, newlines and form-feeds) to be printed visibly. Control characters are printed as ^*X* (control-*x*). The DEL character (octal 0177) is printed as ^?. Non-ASCII characters (with the high bit set) are printed as M-*x*, where *x* is the character specified by the seven low order bits.

When **–v** is specified, **–t** causes each tab to be replaced by **^I**, and **–e** causes a **$** character to be printed at the end of each line (prior to the newline). The **–t** and **–e** options are ignored if **–v** is not specified.

WARNING

Command formats such as

cat file1 file2 >file1

will cause the original data in *file1* to be lost; therefore, take care when using shell special characters.

SEE ALSO

cp(1), pr(1).

NAME

cb – C program beautifier

SYNOPSIS

cb [**–s**] [**–j**] [**–l** leng] [file ...]

DESCRIPTION

Cb reads C programs either from its *file* arguments or from the standard input, and writes them on the standard output with spacing and indentation that displays the structure of the code.

By default, all user newlines are preserved. If the **–s** flag is specified, the code is canonicalized to the style of Kernighan and Ritchie in *The C Programming Language*. The **–j** flag causes split lines to be rejoined. The **–l** flag causes *cb* to split lines that are longer than *leng*.

SEE ALSO

cc(1).

The C Programming Language by B. W. Kernighan and D. M. Ritchie (available from PERQ Systems).

BUGS

Punctuation that is hidden in preprocessor statements will cause indentation errors.

NAME

cd – change working directory

SYNOPSIS

cd [directory]

DESCRIPTION

Cd changes the current working directory to *directory*. If *directory* is not specified, the value of the shell parameter **$HOME** is used as the new working directory. If *directory* specifies a complete path starting with /, ., or .., *directory* becomes the new working directory. If neither case applies, *cd* tries to find the designated directory relative to one of the paths specified by the **$CDPATH** shell variable. **$CDPATH** has the same syntax as, and similar semantics to, the **$PATH** shell variable. *Cd* must have execute (search) permission in *directory*.

Because a new process is created to execute each command, *cd* would be ineffective if it were written as a normal command. Therefore, it is recognized and is internal to the shell.

SEE ALSO

pwd(1), sh(1).
chdir(2).

NAME

cmp – compare two files

SYNOPSIS

cmp [**–l**] [**–s**] file1 file2

DESCRIPTION

Cmp compares *file1* and *file2*. If *file1* is **–**, the standard input is used.

Normally, *cmp* makes no comment if the files are the same. If they differ, it prints the byte and line number at which the difference occurred. It will report if one file is an initial subsequence of the other.

Options:

–l Print the byte number (decimal) and the differing bytes (octal) for each difference.

–s Print nothing for differing files; return exit codes only.

SEE ALSO

diff(1).

DIAGNOSTICS

Exit code 0 is returned for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

NAME

cp, mv – copy or move files

SYNOPSIS

cp file1 [file2 ...] target
mv [**–f**] file1 [file2 ...] target

DESCRIPTION

Cp copies *file1* to *target*. *Mv* moves (i.e., renames) *file1* to *target*. Under no circumstance can *file1* and *target* be the same (take care when using *sh*(1) metacharacters). If *target* is a directory, then one or more files are copied (moved) to that directory. If *target* is a file, its original contents (if any) are destroyed.

If *mv* determines that the mode of *target* forbids writing, it prints the mode (see *chmod*(2)), asks for a response, and reads the standard input for one line. If the line begins with **y**, the *mv* occurs, if permissible; if not, the command exits. When the **–f** option is used, or if the standard input is not a terminal, *mv* will not ask for input.

Only *mv* allows *file1* to be a directory, in which case the directory rename will occur only if the two directories have the same parent; *file1* is renamed *target*.

When using *cp*, if *target* is not an existing file, a new file is created which has the same mode as *file1*; the owner and group of *target* are those of the user. If *target* is a file, copying a file into *target* does not change its mode, owner, or group. The last modification time of *target* (and last access time, if *target* did not exist) and the last access time of *file1* are set to the time the copy was made.

SEE ALSO

rm(1).
chmod(2).

BUGS

If *file1* and *target* lie on different file systems, *mv* must first copy the file and then delete the original.

NAME

cpp – the C language preprocessor

SYNOPSIS

/lib/cpp [option ...] [ifile [ofile]]

DESCRIPTION

Cpp is the C language preprocessor that is invoked as the first pass of any C compilation using the *cc*(1) command. The output of *cpp* will be in a form acceptable as input to the next pass of the C compiler. The preferred way to invoke *cpp* is through the *cc*(1) command, since the functionality of *cpp* may someday be moved elsewhere. The use of *cpp* other than in the framework of *cc* is not suggested.

Cpp optionally accepts two filenames as arguments. *Ifile* and *ofile* are respectively the input and output for the preprocessor. They default to standard input and standard output if not supplied.

The following *options* to *cpp* are recognized:

–P Preprocess the input without producing the line control information used by the next pass of the C compiler.

–C Pass comments along (except those found on *cpp* directive lines). If this option is not specified, *cpp* strips C-style comments.

–U*name*

Remove any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor. The current list of these possibly reserved symbols includes:

operating system:	ibm, gcos, os, tss, unix
hardware:	interdata, pdp11, u370, u3b, u3b5, vax
UNIX system variant:	RES, RT
lint(1):	lint

–D*name*

–D*name=def*

Define *name* as if by a **#define** directive. If no *=def* is given, *name* is defined as 1. The **–D** option has lower precedence than the **–U** option. That is, if the same name is used in both a **–U** option and a **–D** option, the name will be undefined regardless of the order of the options.

–T Force *cpp* to use only the first eight characters of preprocessor symbols for distinguishing different preprocessor names. This

behavior is the same as previous preprocessors with respect to the length of names and is included for backward compatibility.

−I*dir* Search for **#include** files whose names do not begin with / in *dir* first, before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in "" will be searched for first in the directory of the file with the **#include** line, then in directories named in −I options, and last in directories on a standard list. For **#include** files whose names are enclosed in <>, the directory of the file with the **#include** line is not searched. See the description of the **#include** directive below.

Two special names are understood by *cpp*. The name **__LINE__** is defined as the current line number (a decimal integer) as known by *cpp*, and **__FILE__** is defined as the current filename (a C string) as known by *cpp*. They can be used anywhere (including in macros) just as any other defined name.

All *cpp* directives start with lines beginning with #. Any number of blanks and tabs are allowed between the # and the directive. The directives are:

#define *name token-string*
Replace subsequent instances of *name* with *token-string*.

#define *name*(*arg*, ..., *arg*) *token-string*
Replace with *token-string* all subsequent instances of *name* followed by a (, a list of comma-separated sets of tokens, and a). Each occurrence of an *arg* in the *token-string* is replaced by the corresponding set of tokens in the comma-separated list. When a macro with arguments is expanded, the arguments are placed into the expanded *token-string* unchanged. After the entire *token-string* has been expanded, *cpp* re-starts its scan for names to expand at the beginning of the newly created *token-string*. Note that there is no space between *name* and the following (.

#undef *name*
Remove the definition of *name* (if any) from now on.

#include "*filename*"
#include <*filename*>
Include at this point the contents of *filename* (which will then be run through *cpp*). When the <*filename*> notation is used, *filename* is only searched for in the standard places. See the −I option above for more detail.

#line *integer-constant "filename"*
Generate line control information for the next pass of the C compiler. *Integer-constant* is the line number of the next line and *filename* is the file that the lines are read from. If "*filename*" is not given, the current filename is unchanged.

#endif
End a section of lines begun by a test directive (**#if**, **#ifdef**, or **#ifndef**). Each test directive must have a matching **#endif**.

#ifdef *name*
The lines following will appear in the output if and only if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

#ifndef *name*
The lines following will not appear in the output if and only if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

#if *constant-expression*
Lines following will appear in the output if and only if the *constant-expression* evaluates to non-zero. All binary non-assignment C operators, the **?:** operator, and the unary **−**, **!**, and **~** operators are all legal in *constant-expression*. The precedence of the operators is the same as defined by the C language. There is also a unary operator **defined**, which can be used in *constant-expression* in these two forms: **defined (** *name* **)** or **defined** *name*. This allows the utility of **#ifdef** and **#ifndef** in an **#if** directive. Only those operators, integer constants, and names that are known by *cpp* should be used in *constant-expression*. In particular, the **sizeof** operator is not available.

#else Reverse the notion of the test directive that matches this directive. For example, if lines previous to this directive are ignored, following lines will appear in the output.

The test directives and the possible **#else** directives can be nested.

FILES

include:	Searchlist for **#include** files.
/usr/include	standard directory for **#include** files

SEE ALSO

cc(1), m4(1).

DIAGNOSTICS

The error messages produced by *cpp* are intended to be self-explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

NOTES

When newline characters were found in argument lists for macros to be expanded, previous versions of *cpp* put out the newlines as they were found and expanded. The current version of *cpp* replaces these newlines with blanks to alleviate problems that the previous versions had when this occurred.

NAME

csplit – context split

SYNOPSIS

csplit [–s] [–k] [–f prefix] file arg1 [... argn]

DESCRIPTION

Csplit reads *file* and separates it into n+1 sections, defined by the arguments *arg1... argn*. By default, the sections are placed in xx00 ... xx*n* (*n* may not be greater than 99). Each section gets the following piece of *file*:

00: From the start of *file* up to (but not including) the line referenced by *arg1*.

01: From the line referenced by *arg1* up to the line referenced by *arg2*.

.
.
.

n: From the line referenced by *argn* to the end of *file*.

If *file* is –, the standard input is used.

The options to *csplit* are:

–s Suppress the printing of all character counts. *Csplit* normally prints the character count for each file created.

–k Do not delete previously created files if an error occurs. *Csplit* normally removes created files if an error occurs.

–f *prefix* Name the created files *prefix***00 ...** *prefixn*. The default prefix is **xx**.

The arguments (*arg1 ... argn*) to *csplit* can be a combination of the following:

/rexp/ Creates a file for the section from the current line up to (but not including) the line containing the regular expression *rexp*. The current line becomes the line containing *rexp*. This argument may be followed by an optional + or – some number of lines (e.g., **/Page/–5**). See the description of regular expressions in *ed*(1).

%rexp% The same as */rexp/*, except that no file is created for the section.

lnno Creates a file from the current line up to (but not including) *lnno*. The current line becomes *lnno*.

{*num*} Repeat argument. This argument may follow any of the above arguments. If it follows either *rexp* argument, that argument is applied *num* more times. If it follows a *lnno* argument, the file is split every *lnno* lines from that point, *num* times.

All *rexp* type arguments that contain blanks or other characters that are meaningful to the Shell should be enclosed in the appropriate quotes. Regular expressions may not contain embedded newlines.

Csplit does not affect the original file; it is the user's responsibility to remove it if desired.

EXAMPLES

```
csplit -f cobol file '/procedure division/' /par5./ /par16./
```

This example creates four files, **cobol00 ... cobol03**. After editing the "split" files, they can be recombined as follows:

```
cat cobol0[0-3] > file
```

Note that this example overwrites the original file.

```
csplit -k file 100 {99}
```

This example would split the file at every 100 lines, up to 10,000 lines. The **-k** option causes the created files to be retained if there are less than 10,000 lines; however, an error message would still be printed.

```
csplit -k prog.c '%main(%' '/^}/+1' {20}
```

Assuming that **prog.c** follows the normal **C** coding convention of ending routines with a **}** at the beginning of the line, this example will create a file containing each separate **C** routine (up to 21) in **prog.c**.

SEE ALSO

ed(1), sh(1).

DIAGNOSTICS

Error messages should be self-explanatory except for:

arg – out of range

This error means that the given argument did not reference a line between the current position and the end of the file.

NAME

cut – cut out selected fields of each line of a file

SYNOPSIS

cut **–c**list [file1 file2 ...]
cut **–f**list [**–d**char] [**–s**] [file1 file2 ...]

DESCRIPTION

Cut is used to cut out columns from a table or fields from each line of a file. The fields, specified by *list*, can be of fixed length (the **–c** option; for example, character positions as on a punched card) or the length can vary from line to line and be marked with a field delimeter character like *tab* (the **–f** option). *Cut* can be used as a filter; if no *files* are given, the standard input is used.

List is a comma-separated list of integer field numbers (in increasing order). An optional - indicates ranges; for example, **1,4,7**; **1–3,8**; **–5,10** (short for **1–5,10**); or **3–** (third through last field).

Options for *cut* are:

–c*list* The *list* following **–c** (no space) specifies character positions. For example, **–c1–72** would pass the first 72 characters of each line.

–f*list* The *list* following **–f** is a list of fields assumed to be separated in the file by a delimiter character (see the **–d** option). For example, **–f1,7** copies the first and seventh field only. Lines with no field delimiters will be passed through intact (useful for table subheadings), unless **–s** is specified.

–d*char* The character following **–d** is the field delimiter (**–f** option only). Default is *tab*. Space or other characters with special meaning to the shell must be quoted.

–s Suppresses lines with no delimiter characters in case of **–f** option. Unless specified, lines with no delimiters will be passed through untouched.

Either the **–c** or **–f** option must be specified.

HINTS

Use *grep*(1) to make horizontal "cuts" (by context) through a file, or *paste*(1) to put files together column-wise (i.e., horizontally). To reorder columns in a table, use *cut* and *paste*.

EXAMPLES

cut −d: −f1,5 /etc/passwd
maps of user IDs to names

name= `who am i | cut −f1 −d" "`
sets **name** to current login name.

DIAGNOSTICS

line too long
A line can have no more than 1023 characters or fields.

bad list for c/f option
Missing **−c** or **−f** option or incorrectly specified *list*. No error occurs if a line has fewer fields than the *list* calls for.

no fields
The *list* is empty.

SEE ALSO

grep(1), paste(1).

NAME

diff − differential file comparator

SYNOPSIS

diff [**−efbh**] file1 file2

DESCRIPTION

Diff compares two files and outputs the lines in the files that must be changed to make both files identical. If *file1* or *file2* is **−**, the standard input is used. If *file1* or file2 is a directory, a file in that directory with the same name as the other file is used.

The normal output from *diff* contains lines of these forms:

n1 **a** *n3,n4*
n1,n2 **d** *n3*
n1,n2 **c** *n3,n4*

n1, *n2*, ... are line numbers. The numbers to the left of the letters refer to lines in *file1*, and the numbers to the right refer to *file2*. These lines resemble *ed* commands; they show how to convert *file1* to *file2*. By exchanging **a** for **d** (and vice versa) and reading backward, it is possible to convert *file2* to *file1*. As in *ed*, identical pairs of lines (for example, if *n1* = *n2* or *n3* = *n4*) are abbreviated as a single number.

Each output line as shown above is followed by all lines that are affected. Lines in *file1* are flagged by **>**, and lines in *file2* are flagged by **<**.

The **−b** option causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to be considered equal.

The **−e** option produces a script of **a**, **c**, and **d** commands for the editor *ed* that will convert *file1* to match *file2*. The **−f** option produces a similar script, not usable by *ed*, that would convert *file2* to match *file1*.

The **−h** option does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length. Options **−e** and **−f** are unavailable with **−h**.

By using the **−e** option, only an ancestral file ($1) and a chain of version-to-version *ed* scripts ($2, $3, ...) made by *diff* are needed to maintain multiple versions of a file. The following shell program would generate the "latest version" of the file on the standard output:

```
(shift; cat $*; echo '1,$p') | ed − $1
```

Except in rare circumstances, *diff* finds the smallest sufficient set of file differences.

FILES

/tmp/d?????
/usr/lib/diffh for **−h**

SEE ALSO

cmp(1), ed(1).

DIAGNOSTICS

Exit status is 0 for no differences, 1 for some differences, 2 for trouble.

BUGS

Editing scripts produced under the **−e** or **−f** option are naive about creating lines consisting of a single period (.).

WARNINGS

Missing newline at end of file X

indicates that the last line of file X did not have a newline. If only one of the files is missing the newline on the last line, the lines will be flagged and output, although the output will seem to indicate that the lines are the same.

NAME

echo – echo arguments

SYNOPSIS

echo [arg] ...

DESCRIPTION

Echo writes its arguments, separated by blanks and terminated by a newline, on the standard output. It also understands C-like escape conventions (note: beware of conflicts with the shell's use of \):

\b	backspace
\c	print line without newline
\f	form-feed
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
****	backslash
*****n*	the 8-bit character whose ASCII code is the 1-, 2- or 3-digit octal number *n*, which must start with a zero.

Echo is useful for producing diagnostics in command files and for sending known data into a pipe.

SEE ALSO

sh(1).

NAME

ed, red – text editor

SYNOPSIS

ed [–] [file]

red [–] [file]

DESCRIPTION

Ed is the standard text editor. If the *file* argument is given, *ed* reads the file into its buffer so that it can be edited (see the *e* command below).

Ed operates on a copy of the file it is editing. Changes made to the copy have no effect on the file until a *w* (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer*. There is only one buffer.

Red is a restricted version of *ed*. It only allows the editing of files in the current directory, and prohibits execution of shell commands via !*shellcommand*. Attempts to bypass these restrictions result in an error message (*restricted shell*).

Specifying the optional – on the command line suppresses the printing of character counts by the *e*, *r*, and *w* commands, the printing of diagnostics from *e* and *q* commands, and the printing of the ! prompt after a !*shellcommand*.

Commands to *ed* have a simple and regular structure: zero, one, or two *addresses* followed by a single-character *command*, possibly followed by parameters to that command. The addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that they can often be omitted. In general, only one command may appear on a line.

Certain commands allow the input of text. This text will be placed in the appropriate place in the buffer. While *ed* is accepting text, it is said to be in *input mode*. In this mode, *no* commands are recognized; all input is merely collected. Input mode is left by typing a period (.) alone at the beginning of a line.

Ed supports a limited form of *regular expression* notation. Regular expressions are used in addresses to specify lines, and in some commands (e.g., *s*) to specify portions of a line that are to be substituted. A regular expression (RE) specifies a set of character strings. A member of this set of strings is said to be *matched* by the RE. The REs allowed by *ed* are constructed as follows:

The following *one-character RE*s match a *single* character:

1.1 An ordinary character (*not* one of those discussed in 1.2 below) is a one-character RE that matches itself.

1.2 A backslash (\) followed by any special character is a one-character RE that matches the special character itself. The special characters are:

a. ., *, [, and \ (period, asterisk, left square bracket, and backslash, respectively); which are always special, *except* when they appear within square brackets ([]; see 1.4 below).

b. ^ (caret or circumflex), which is special at the *beginning* of an *entire* RE (see 3.1 and 3.2 below), or when it immediately follows the left of a pair of square brackets ([]; see 1.4 below).

c. $ (currency symbol), which is special at the *end* of an entire RE (see 3.2 below).

d. The character used to bound (i.e., delimit) an entire RE, which is special for that RE (for example, note the way slash (/) is used in the *g* command below.)

1.3 A period (.) is a one-character RE that matches any character except newline.

1.4 A non-empty string of characters enclosed in square brackets ([]) is a one-character RE that matches *any one* character in that string. If, however, the first character of the string is a caret (^), the one-character RE matches any character *except* newline and the remaining characters in the string. The ^ has this special meaning *only* if it is the first character enclosed in the square brackets. The minus (−) may be used to indicate a range of consecutive ASCII characters; for example, **[0−9]** is equivalent to **[0123456789]**. The − loses this special meaning if it occurs first (after an initial ^, if any) or last in the string. The right square bracket (]) does not terminate such a string when it is the first character within it (after an initial ^, if any); e.g., **[]a−f]** matches either a right square bracket (]) or one of the letters a through f inclusive. The characters listed in 1.2.a stand for themselves within such a string of characters.

The following rules may be used to construct *RE*s from one-character REs:

2.1 A one-character RE is a RE that matches whatever the one-character RE matches.

2.2 A one-character RE followed by an asterisk (*) is a RE that matches *zero or more* occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character RE followed by \{*m*\}, \{*m*,\}, or \{*m*,*n*\} is a RE that matches a *range* of occurrences of the one-character RE. The values of *m* and *n* must be non-negative integers less than 256. \{*m*\} matches *exactly m* occurrences; \{*m*,\} matches *at least m* occurrences; \{*m*,*n*\} matches *any number* of occurrences *between m* and *n* inclusive. Whenever a choice exists, the RE matches as many occurrences as possible.

2.4 The concatenation of REs produces a RE that matches the concatenation of the strings matched by each component of the RE.

2.5 A RE enclosed between the character sequences \(and \) is a RE that matches whatever the unadorned RE matches.

2.6 The expression *n* matches the same string of characters as was matched by an expression enclosed between \(and \) *earlier* in the same RE. The expression specified is that beginning with the *n*-th occurrence of \(counting from the left. For example, the expression ^\(.*\)\1$ matches a line consisting of two repeated appearances of the same string.

Finally, an *entire RE* may be constrained to match only an initial segment or final segment of a line (or both):

3.1 A caret (^) at the beginning of an entire RE constrains that RE to match an *initial* segment of a line.

3.2 A currency symbol ($) at the end of an entire RE constrains that RE to match a *final* segment of a line.

The construction ^*entire RE*$ constrains the entire RE to match the entire line.

The null RE (e.g., //) is equivalent to the last RE encountered. See also the last paragraph before *FILES* below.

To understand addressing in *ed*, it is necessary to know that there is a *current line* at all times. Generally, the current line is the last line affected by a command; the exact effect each command has on the current line is discussed under its description. *Addresses* are constructed as follows:

1. The character . addresses the current line.

2. The character **$** addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. '*x* addresses the line marked with the mark named *x*, which must be a lower-case letter. Lines are marked with the *k* command described below.
5. A RE enclosed by slashes (**/**) addresses the first line that matches the RE found by searching *forward* starting from the line *following* the current line. If necessary, the search wraps around to the beginning of the buffer and continues up to and including the current line, so that the entire buffer is searched. See also the last paragraph before *FILES* below.
6. A RE enclosed in question marks (**?**) addresses the first line that matches the RE found by searching *backward* from the line *preceding* the current line. If necessary, the search wraps around to the end of the buffer and continues up to and including the current line. See also the last paragraph before *FILES* below.
7. An address followed by a plus sign (**+**) or a minus sign (**−**) followed by a decimal number specifies that address plus (or minus) the indicated number of lines. The plus sign may be omitted.
8. If an address begins with **+** or **−**, the addition or subtraction is taken with respect to the current line; e.g, **−5** is understood to mean **.−5**.
9. If an address ends with **+** or **−**, without a following number, 1 is added to or subtracted from the address. As a consequence of this rule and of rule 8, the address **−** refers to the line preceding the current line. (To maintain compatibility with earlier versions of the editor, the character **^** in addresses is equivalent to **−**.) Moreover, multiple **+** and **−** characters have a cumulative effect, so **−−** refers to the current line less 2.
10. For convenience, a comma by itself (**,**) stands for the address pair **1,$**, while a semicolon by itself (**;**) stands for the pair **.,$**.

Commands may require zero, one, or two addresses. Commands that require zero addresses regard the presence of an address as an error. Commands that accept one or two addresses assume default addresses when an insufficient number of addresses is given; if more addresses are given than such a command requires, the last one(s) are used.

Typically, addresses are separated from each other by a comma (,). They may also be separated by a semicolon (;). If a semicolon is used, the current line (.) is set to the first address, and only then is the second address calculated. This feature can be used to determine the starting line for forward and backward searches (see rules 5 and 6 above). The second address of any two-address sequence must correspond to a line that follows the line corresponding to the first address.

It is generally illegal for more than one command to appear on a line. However, any command (except *e*, *f*, *r*, or *w*) may be suffixed by **l**, **n** or **p**, in which case the current line is either listed, numbered or printed, respectively, as discussed below under the *l*, *n* and *p* commands.

In the following list of *ed* commands, the default addresses are shown in parentheses. The parentheses are *not* part of the address; they show that the given addresses are the default.

(.)a
<text>
.

The *a*ppend command reads the given text and appends it after the addressed line; . is left at the last inserted line, or, if no lines were inserted, at the addressed line. Address 0 is legal for this command: it causes the "appended" text to be placed at the beginning of the buffer. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

(.)c
<text>
.

The *c*hange command deletes the addressed lines, then accepts input text that replaces these lines. . is left at the last line inserted, or, if no lines were inserted, at the first line that was not deleted.

(.,.)d

The *d*elete command deletes the addressed lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end of the buffer, the new last line becomes the current line.

e *file*

The *e*dit command deletes the entire contents of the buffer and then reads in the named file; . is set to the last line of the buffer.

If no filename is given, the currently-remembered filename, if any, is used (see the *f* command). The number of characters read is typed. *File* is remembered for possible use as a default filename in subsequent *e*, *r*, and *w* commands. If *file* is replaced by !, the rest of the line is taken to be a shell (*sh*(1)) command whose output is to be read. Such a shell command is *not* remembered as the current filename. This command warns the user if any changes have been made to the buffer since the last *w* command. See also *DIAGNOSTICS* below.

E *file*

The *E*dit command is like *e*, except that the editor does not check to see if any changes have been made to the buffer since the last *w* command.

f *file*

If *file* is given, the *f*ile-name command changes the currently-remembered filename to *file*. If *file* is not given, it prints the currently-remembered filename.

(1,$)g/*RE/ command list*

The *g*lobal command first marks every line that matches the given RE. Then, the given *command list* is executed with . set to each marked line successively. A single command or the first of a list of commands appears on the same line as the *g* command. If a list of commands is given, all lines of the list of commands except the last line must end with a \. The *a*, *i*, and *c* commands and associated input are permitted. The *g*, *G*, *v*, and *V* commands are not permitted in the *command list*. The . terminating input mode may be omitted if it would be the last line of the *command list*. An empty *command list* is equivalent to the *p* command. See also *BUGS* and the last paragraph before *FILES* below.

(1,$)G/*RE/*

The interactive *G*lobal command first marks every line that matches the given RE. Then, for each marked line, the line is printed, **and any** *one* command (other than *a*, *c*, *i*, *g*, *G*, *v*, or *V*) may be input and is executed. After the command is executed, the next marked line is printed, and so on. A newline acts as a null command; an **&** causes the re-execution of the last command executed within the current *G*. Note that the commands input to the *G* command may address and affect *any* lines in the buffer. The *G* command can be terminated by an interrupt signal (ASCII

DEL or BREAK).

h

The *h*elp command gives a short error message that explains the reason for the most recent **?** diagnostic.

H

The *H*elp command causes *ed* to enter a mode in which error messages are printed for all subsequent **?** diagnostics. It will also explain the previous **?** if there was one. The *H* command alternately turns this mode on and off; it is initially off.

(.)i
<text>

The *i*nsert command inserts the given text before the addressed line; . is left at the last inserted line, or, if no lines were inserted, at the addressed line. The difference between the *i* and *a* commands is that address 0 is not legal for *i*. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

(.,.+1)j

The *j*oin command joins contiguous lines by removing the appropriate newline characters. If only one address is given, this command does nothing.

(.)k*x*

The mar*k* command marks the addressed line with name *x*, which must be a lower-case letter. The address *'x* will address the marked line; . is unchanged.

(.,.)l

The *l*ist command prints the addressed lines. Some non-printing characters (e.g., *tab, backspace*) are represented by (hopefully) mnemonic overstrikes; all other non-printing characters are printed in octal. Long lines are folded. An *l* command may be appended to any other command except for *e*, *f*, *r*, and *w*. . is left at the last line listed.

(.,.)m*a*

The *m*ove command repositions the addressed line(s) after the line addressed by *a*. Address 0 is legal for *a* and causes the addressed line(s) to be moved to the beginning of the file. It is an error if address *a* falls within the range of moved lines. . is left at the last

line moved.

(.,.)n

The *n*umber command prints the addressed lines, preceding each line by its line number and a tab character. . is left at the last line printed. The *n* command may be appended to any other command except for *e*, *f*, *r*, and *w*.

(.,.)p

The *p*rint command prints the addressed lines. . is left at the last line printed. The *p* command may be appended to any other command except for *e*, *f*, *r*, and *w*. For example, *dp* deletes the current line and prints the new current line.

P

The *P*rompt command causes the editor to prompt with a * for all subsequent commands. The *P* command alternately turns this mode on and off; it is initially off.

q

The *q*uit command causes *ed* to exit. No automatic write of a file is done, but the user is warned if the buffer has been modified since the last *w* command (see *DIAGNOSTICS* below).

Q

The editor exits without checking if changes have been made in the buffer since the last *w* command.

(.)r *file*

The *r*ead command reads in the given file after the addressed line. If no *file* is given, the currently-remembered filename, if any, is used (see the *e* and *f* commands). The currently-remembered filename is *not* changed unless *file* is the very first filename mentioned since *ed* was invoked. Address 0 is legal for *r*, and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed; . is set to the last line read in. If *file* is replaced by !, the rest of the line is taken to be a shell (*sh*(1)) command whose output is to be read. For example, "$r !ls" appends the current directory to the end of the file being edited. Such a shell command is *not* remembered as the current filename.

(.,.)s/*RE***/***replacement***/** or
(.,.)s/*RE***/***replacement***/g**

The *s*ubstitute command searches each addressed line for an

occurrence of the specified RE. In each line in which a match is found, all (non-overlapped) matched strings are replaced by the *replacement* if the global replacement indicator **g** appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. It is an error for the substitution to fail on *all* addressed lines. Any character other than space or newline may be used instead of **/** to delimit the RE and the *replacement*. . is left at the last line on which a substitution occurred. See also the last paragraph before *FILES* below.

An ampersand (**&**) appearing in the *replacement* is replaced by the string matching the RE on the current line. The special meaning of **&** in this context may be suppressed by preceding it by ****. More generally, the characters *****n*, where *n* is a digit, are replaced by the text matched by the *n*-th regular subexpression of the specified RE enclosed between **\(** and **\)**. When nested parenthesized subexpressions are present, *n* is determined by counting occurrences of **\(** starting from the left. When the character **%** is the only character in the *replacement*, the *replacement* used in the most recent substitute command is used as the *replacement* in the current substitute command. The **%** loses its special meaning when it is in a replacement string of more than one character or is preceded by a ****.

A line may be split by substituting a newline character into it. The newline in the *replacement* must be preceded by ****. Such substitution cannot be done as part of a *g* or *v* command list.

(.,.)t*a*

This command acts just like the *m* command, except that a *copy* of the addressed lines is placed after address *a* (which may be 0). . is left at the last line of the copy.

u

The *u*ndo command nullifies the effect of the most recent command that modified anything in the buffer, namely the most recent *a*, *c*, *d*, *g*, *i*, *j*, *m*, *r*, *s*, *t*, *v*, *G*, or *V* command.

(1,$)v/*RE/command list*

This command is the same as the global command *g* except that the *command list* is executed with . initially set to every line that does *not* match the RE.

(1,$)V/*RE***/**

This command is the same as the interactive global command *G* except that the lines that are marked during the first step are those that do *not* match the RE.

(1,$)w *file*

The write command writes the addressed lines into the named file. If the file does not exist, it is created with mode 666 (readable and writable by everyone), unless your *umask* setting dictates otherwise (see *sh*(1)). The currently-remembered filename is *not* changed unless *file* is the very first filename mentioned since *ed* was invoked. If no filename is given, the currently-remembered filename, if any, is used (see *e* and *f* commands). . is unchanged. If the command is successful, the number of characters written is typed. If *file* is replaced by **!**, the rest of the line is taken to be a shell (*sh*(1)) command whose standard input is the addressed lines. Such a shell command is *not* remembered as the current filename.

($)=

The line number of the addressed line is typed. . is unchanged by this command.

!*shellcommand*

The remainder of the line after the **!** is sent to the UNIX shell (*sh*(1)) to be interpreted as a command. Within the text of that command, the unescaped character **%** is replaced with the remembered filename; if a **!** appears as the first character of the shell command, it is replaced with the text of the previous shell command. Thus, **!!** will repeat the last shell command. If any expansion is performed, the expanded line is echoed. . is unchanged.

(.+1)<newline>

An address alone on a line causes the addressed line to be printed. A newline alone is equivalent to **.+1p**; it is useful for stepping forward through the buffer. . is changed to the line printed.

If an interrupt signal (ASCII DEL or BREAK) is sent, *ed* prints a **?** and returns to *its* command level.

Ed has the following size limitations:

512 characters per line;

256 characters per global command list;

64 characters per filename;

and 128K characters in the buffer.

The limit on the number of lines depends on the amount of user memory; each line takes 1 word.

When reading a file, *ed* discards ASCII NULL characters and all characters after the last newline. Files (e.g., **a.out**) that contain non-ASCII characters (bit 8 set) cannot be edited by *ed*.

If the closing delimiter of a RE or of a replacement string (e.g., /) would be the last character before a newline, that delimiter may be omitted, in which case the addressed line is printed. The following pairs of commands are equivalent:

s/s1/s2	s/s1/s2/p
g/s1	g/s1/p
?s1	?s1?

FILES

/tmp/e#	temporary; # is the process number.
ed.hup	work is saved here if the terminal is hung up.

DIAGNOSTICS

?	for command errors.
?*file*	for an inaccessible file.
	(use the *h*elp and *H*elp commands for detailed explanations).

If changes have been made in the buffer since the last *w* command that wrote the entire buffer, *ed* warns the user if an attempt is made to destroy *ed*'s buffer via the *e* or *q* commands: it prints ? and continues editing. A second *e* or *q* command at this point will take effect. The – command-line option inhibits this feature.

SEE ALSO

grep(1), sh(1).
The document "Tutorial - Text Editor" in this manual.

LIMITATIONS

A *!* command cannot be subject to a *g* or a *v* command.
The *!* command and the ! escape from the *e*, *r*, and *w* commands cannot be used if the editor is invoked from a restricted shell (see *sh*(1)).
The sequence \n in a RE does not match a newline character.
The *l* command mishandles DEL.

Characters are masked to 7 bits on input.

NAME

expr – evaluate arguments as an expression

SYNOPSIS

expr arguments

DESCRIPTION

Expr evaluates *arguments* as an expression. After evaluation, the result is written on the standard output. Terms of the expression must be separated by blank spaces. Note that **0**, not the null string, is returned to indicate a zero value.

Characters that are special to the shell must be escaped (preceded by \). Strings containing blanks or other special characters should be quoted. Integer-valued arguments may be preceded by a unary minus sign. Internally, integers are treated as 32-bit, 2s complement numbers.

The operators and keywords for expressions are listed below. Characters that need to be escaped are preceded by \. The list is in order of increasing precedence, with equal precedence operators grouped within **{}** symbols.

expr **\|** *expr*
: returns the first *expr* if it is neither null nor **0**, otherwise returns the second *expr*.

expr **\&** *expr*
: returns the first *expr* if neither *expr* is null or **0**, otherwise returns **0**.

expr **{ =, \>, \>=, \<, \<=, != }** *expr*
: returns the result of an integer comparison if both arguments are integers, otherwise returns the result of a lexical comparison.

expr **{ +, – }** *expr*
: returns the sum or difference of integer-valued arguments.

expr **{ *, /, % }** *expr*
: returns the product, quotient, or remainder of integer-valued arguments.

expr **:** *expr*
: compares the first argument with the second argument, which must be a regular expression. Regular expression syntax is the same as that for *ed*(1), except that all patterns are "anchored" (i.e., begin with ^) and, therefore, ^ is not a special character. Normally, the matching operator returns the number of characters

matched (**0** on failure). Alternatively, the **\(...\)** pattern symbols can be used to return a portion of the first argument.

EXAMPLES

1. a=`expr $a + 1`

   adds 1 to the shell variable **a**.

2. # For $a equal to either "/usr/abc/file" or just # "file":

   expr $a : '.*/\(.*\)' \| $a

   returns the last segment of a pathname (i.e., file). Watch out for **/** alone as an argument; *expr* will take it as the division operator (see BUGS below).

3. # A better representation of example 2:

   expr //$a : '.*/\(.*\)'

   The addition of the **//** characters eliminates any ambiguity about the division operator and simplifies the whole expression.

4. expr $VAR : '.*'

   returns the number of characters in **$VAR**.

SEE ALSO

ed(1), sh(1).

EXIT CODE

As a side effect of expression evaluation, *expr* returns the following exit values:

0	if the expression is neither null nor **0**
1	if the expression *is* null or **0**
2	for invalid expressions.

DIAGNOSTICS

syntax error for operator/operand errors

non-numeric argument

if arithmetic is attempted on such a string

BUGS

After argument processing by the shell, *expr* cannot tell the difference between an operator and an operand except by the value. If **$a** is an **=**, the command

expr $a = '='

looks like

```
expr = = =
```

as the arguments are passed to *expr* (and they will all be taken as the = operator). The following works:

```
expr X$a = X=
```

NAME

grep, egrep, fgrep – search a file for a pattern

SYNOPSIS

grep [options] expression [files]

egrep [options] [expression] [files]

fgrep [options] [strings] [files]

DESCRIPTION

Commands of the *grep* family search the input *files* for lines matching a pattern. The standard input is used if no *files* are specified. Normally, each line found is copied to the standard output.

Grep patterns are limited regular *expressions* in the style of *ed*(1); it uses a compact non-deterministic algorithm. *Egrep* patterns are full regular *expressions*; it uses a fast deterministic algorithm that sometimes needs exponential space. *Fgrep* patterns are fixed *strings*; it is fast and compact.

The following *options* are recognized:

–v Print all lines but those matching expressions.

–x (Exact) Print only lines that are matched in their entirety (*fgrep* only).

–c Print only a count of matching lines.

–i Ignore upper/lower case distinction during comparisons.

–l List only the names of files with matching lines, separated by newlines. Each file will only be listed once.

–n Precede each line by its relative line number in the file.

–b Precede each line by the block number on which it is found. This is sometimes useful in locating disk block numbers by context.

–s Suppress error messages for nonexistent or unreadable files (*grep* only).

–e *expression*
Same as a simple *expression* argument, but useful when the *expression* begins with a – (*egrep* and *fgrep* only).

–f *file*
Read the regular *expression* (*egrep*) or list of *strings* (*fgrep*) from *file*.

In all cases, the filename is output if there is more than one input file. Care should be taken when using the characters **$**, *****, **[**, **^**, **|**, **(**, **)**, and **** in *expression* because they are also meaningful to the shell. It is safest to enclose the entire *expression* argument in single quotes (**''**).

Fgrep searches for lines that contain one of the *strings* separated by newlines.

Egrep accepts regular expressions as in *ed*(1), except for \(and \), and with the addition of:

1. A regular expression followed by + matches one or more occurrences of the regular expression.
2. A regular expression followed by ? matches 0 or 1 occurrences of the regular expression.
3. Two regular expressions separated by | or by a newline match strings that are matched by either.
4. A regular expression may be enclosed in parentheses () for grouping.

The order of precedence of operators is [], then * ? +, then concatenation, then | and newline.

SEE ALSO

ed(1), sh(1).

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, and 2 for syntax errors or inaccessible files (even if matches were found).

BUGS

Ideally there should be only one *grep*, but there is not a single algorithm that spans a wide enough range of space-time tradeoffs.

Lines are limited to BUFSIZ characters; longer lines are truncated. **BUFSIZ** is defined in **/usr/include/stdio.h**.

Egrep does not recognize ranges, such as **[a−z]**, in character classes.

If there is a line with embedded nulls, *grep* will only match up to the first null; if it matches, the entire line is printed.

NAME

kill – terminate a process

SYNOPSIS

kill [–signo] PID ...

DESCRIPTION

Kill sends signal 15 (terminate) to the processes specified by the process IDs *PID*. This will normally kill processes that do not catch or ignore the signal. The process number of each asynchronous process started with **&** is reported by the shell (unless more than one process is started in a pipeline, in which case the number of the last process in the pipeline is reported). Process numbers can also be found by using the Accent *Systat* command.

The details of the kill are described in *kill*(2). For example, if process number 0 is specified, all processes in the process group are signaled.

The killed process must belong to the current user unless he is super-user.

If a signal number preceded by – is given as first argument, that signal is sent instead of terminate (see *signal*(2)). In particular, "kill –9 ..." is a sure kill.

SEE ALSO

sh(1).
kill(2), signal(2) in the *UNIX System Programmer Reference Manual.*
The description of Systat in the "User Facilities" document in the *Accent User's Manual.*

NAME

ld – link editor for PERQ C object files

SYNOPSIS

ld [options] filename

DESCRIPTION

Ld combines several object files into one, performs relocation, resolves external symbols, and supports symbol table information for symbolic debugging. In the simplest case, the names of several object programs are given, and *ld* combines them, producing an object module that can either be executed or used as input for a subsequent *ld* run.

The output of *ld* is left in **a.out**. By default, this file is executable if no errors occurred during the load.

The following options are recognized by *ld:*

–l*x* Read in library file **lib:lib***x***.a**.

–o *outfile*
Produce an output object file named *outfile*. The name of the default object file is **a.out**.

–x Do not preserve local (non-**.globl**) symbols in the output symbol table; enter external and static symbols only. This option saves some space in the output file.

–m Create a library as the output file. The output file will not be executable, but will be suitable for subsequent link runs.

@filename
Specify indirect files by *@filename*. These files should be lists of filenames to be included in the link. Comments may be included by starting lines with an exclamation mark (!).

–g*mainsymbolname*
Enter a program at a point other than **main** by specifying **–g***mainsymbolname*.

–s*startsymbolname*
Give a linked object file a start point other than **crt0** by specifying **–s***startsymbolname*. This flag should only be used by those who understand its effects.

–v Report on files as they are read, and on the size of the output file.

–L Save all local symbols in the output symbol table, including those starting with "L" and "‘", which are compiler generated.

-help Produce helpful text describing the switches.

-M Output a message for each multiply-defined external definition.

FILES

lib:	library search list. If undefined:
/lib/lib*x*.a	libraries
/usr/lib/lib*x*.a	libraries
a.out	output file

SEE ALSO

~~as(1), cc(1).~~ asm(1), cpp(1), pcc(1).
exec(2).
The documents "PERQ C Programming" and "C System Interfaces" in the *Accent Languages Manual*, provided with the C compiler.

NAME

lex – generate programs for simple lexical tasks

SYNOPSIS

lex [**–rctvn**] [file] ...

DESCRIPTION

Lex generates programs to be used in simple lexical analysis of text.

The input *files* (standard input default) contain strings and expressions to be searched for, and C text to be executed when the strings are found.

Lex generates a C program in the file **lex.yy.c**. This program, when loaded with the library **lib:libl.a**, reads the input and searches for the strings specified in *lex*'s *file* arguments. When a match is found, the corresponding C text is executed. The input that matched is written into an external character array *yytext*. Input that does not match any of the strings is copied intact to the output. Matching is done in order of the strings in the file.

The strings given in the *files* may contain square brackets to indicate character classes. For example, **[abx–z]** indicates the characters **a**, **b**, **x**, **y**, and **z**. The operators *****, **+**, and **?** mean respectively any non-negative number of, any positive number of, and either zero or one occurrences of the previous character or character class. Thus **[a–zA–Z]+** matches a string of one or more letters. The character **.** matches any one ASCII character except newline. Parentheses for grouping and the vertical bar (**|**) for alternation may also be used in the strings.

The notation *r*{*d*,*e*} in a string indicates between *d* and *e* instances of regular expression *r*. This is of higher precedence than **|**, but lower than *****, **?**, **+**, and concatenation.

The character **^** at the beginning of an expression permits a successful match only immediately after a newline, and the character **$** at the end of an expression requires a trailing newline. The character **/** in an expression indicates trailing context; only the part of the expression up to the slash is returned in *yytext*, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within " symbols or preceded by ****.

Three subroutines defined as macros are expected: **input()** to read a character; **unput(***c***)** to replace a character read; and **output(***c***)** to place an output character. These are defined in **libl.a** in terms of the standard streams, but can be redefined. The program generated is named **yylex()**, and the

library contains a **main()** that calls it. The action **REJECT** on the right side of the rule causes this match to be rejected and the next suitable match executed.

In the *files*, all rules should come after a line containing **&&**, as in YACC. Lines preceding the **&&** that begin with a non-blank character are string definitions; the first string on the line is assigned the contents of the rest of the line. The string's value can be recalled by surrounding it with curly brackets (**{}**). Note that the brackets do not imply parentheses; only the string substitution is done. Any line beginning with a blank is assumed to contain C text and is copied; if it precedes the **&&**, it is copied into the external definition area of the **lex.yy.c** file.

The external names generated by *lex* all begin with the prefix **yy** or **YY**.

Lex accepts the following flags. The flags must appear on the command line before any filenames.

−c indicates C actions. This is the default.

−t causes the normal contents of **lex.yy.c** to be written to the standard output instead.

−v provides a one-line summary of statistics about the program generated.

−n will not print out the **−v** summary.

Multiple *files* are treated as a single file. If no files are specified, the standard input is used.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

%p *n* number of positions is *n* (default 2000)

%n *n* number of states is *n* (500)

%t *n* number of parse tree nodes is *n* (1000)

%a *n* number of transitions is *n* (3000)

The use of one or more of the above automatically implies the **−v** option, unless the **−n** option is used.

EXAMPLE

The following example is a lexical analyzer for a fragment of a C-like language with octal and decimal integers, alphabetic names, 'if' statements, and + and ++ operators. The example recognizes these constructs and, when they are found, prints a suitable message. It also recognizes and

ignores comments delimited by /* and */.

```
D       [0-9]
%%
if      printf("IF statement\n");
[a-z]+  printf("tag, value %s\n",yytext);
0{D}+   printf("octal number %s\n",yytext);
{D}+    printf("decimal number %s\n",yytext);
"++"    printf("unary op\n");
"+"     printf("binary op\n");
"/*"    {       loop:
                while (input() != '*');
                switch (input())
                        {
                        case '/': break;
                        case '*': unput('*');
                        default: go to loop;
                        }
                }
```

SEE ALSO

yacc(1).
malloc(3C).
The document "Lexical Analyzer Generator (lex)" in this manual.

NAME

lint – a C program checker

SYNOPSIS

lint [option] ... file ...

DESCRIPTION

Lint attempts to detect features of C program files that are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compilers.

Among the things that *lint* currently detects are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose values are constant. Moreover, the usage of functions is checked to find functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but not returned.

Files whose names end with **.c** are taken to be C source files. Files whose names end with **.ln** are taken to be the result of an earlier invocation of *lint* with either the **–c** or the **–o** option used. The **.ln** files are analogous to **.o** (object) files produced by the *cc*(1) command when given a **.c** file as input. Files with other suffixes are warned about and ignored.

Lint will take all the **.c**, **.ln**, and **llib-l***x***.ln** (specified by **–l***x*; see below) files and process them in their command line order. By default, *lint* appends the standard C lint library (**llib-lc.ln**) to the end of the list of files. However, if the **–p** option is used, the portable C lint library (**llib-port.ln**) is appended instead. When the **–c** option is not used, the second pass of *lint* checks the list of input files for mutual compatibility. When the **–c** option is used, **.ln** and **llib-l***x***.ln** files are ignored.

Any number of *lint* options may be used, in any order, intermixed with filename arguments. The following options are used to suppress certain kinds of complaints:

–a Suppress complaints about assignments of long values to variables that are not long.

–b Suppress complaints about **break** statements that cannot be reached. Programs produced by *lex* or *yacc* often result in many such complaints.

–u Suppress complaints about functions and external variables used and not defined, or defined and not used. This option is useful

for running *lint* on a subset of files of a larger program.

−v Suppress complaints about unused arguments in functions.

−x Do not report variables referred to by external declarations but never used.

The following options alter *lint*'s behavior:

−h Do not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

−l*x* Include additional lint library **llib-l***x***.ln**. This argument does not suppress the default use of **llib-lc.ln**. These lint libraries must be in the assumed directory. This option can be used to reference local lint libraries and is useful in the development of multi-file projects.

−n Do not check compatibility against either the standard or the portable lint library.

−p Attempt to check portability to other dialects (IBM and GCOS) of C. Along with stricter checking, this option causes all non-external names to be truncated to eight characters and all external names to be truncated to six characters and one case.

−c Produce a **.ln** file for every **.c** file on the command line. These **.ln** files are the product of *lint*'s first pass only, and are not checked for inter-function compatibility.

−o lib Create a lint library with the name **llib-l***lib***.ln**. The lint library produced is the input that is given to *lint*'s second pass. The **−o** option simply causes this file to be saved in the named lint library. The **−x** option should be used to produce a **llib-l***lib***.ln** without extraneous messages. The **−v** option is useful if the source file(s) for the lint library are just external interfaces (for example, the way the file **llib-lc** is written). These option settings are also available through the use of "lint comments" (see below). The **−c** option nullifies any use of the **−o** option.

The **−D**, **−U**, and **−I** options of *cpp*(1) and the **−g** and **−O** options of *cc*(1) are also recognized as separate arguments. The **−g** and **−O** options are ignored, but, by recognizing these options, *lint*'s behavior is closer to that of the *cc*(1) command. Other options are warned about and ignored.

The pre-processor symbol "lint" is defined to allow certain questionable code to be altered or removed for *lint*. Therefore, the symbol "lint"

should be considered a reserved word for all code that is to be checked by *lint*.

Certain conventional comments in the C source will change the behavior of *lint*:

/*NOTREACHED*/
stops comments about unreachable code. This comment is typically placed just after calls to functions like *exit*(2).

/*VARARGS*n*/
suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/*ARGSUSED*/
turns on the **-v** option for the next function.

/*LINTLIBRARY*/
at the beginning of a file shuts off complaints about unused functions and function arguments in this file. This is equivalent to using the **-v** and **-x** options.

Lint produces its first output on a per-source-file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, if the **-c** option is not used, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source filename will be printed followed by a question mark.

The **-c** and the **-o** options allow for incremental use of *lint* on a set of C source files. Generally, *lint* is invoked once for each source file with the **-c** option. Each of these invocations produces a **.ln** file that corresponds to the **.c** file, and prints all messages that are about that source file. After all the source files have been separately run through *lint*, it is invoked once more (without the **-c** option), listing all the **.ln** files with the needed **-l***x* options. This prints all the inter-file inconsistencies. This procedure works well with *make*(1); it allows *make* to be used to *lint* only the source files that have been modified since the last time the set of source files were run through *lint*.

FILES

If the searchlist **lib:** is undefined, the directory **/lib** is used in all cases.

lib:	the directory where the lint libraries specified by the *−lx* option must exist
lib:lint[12]	first and second passes
lib:llib-lc.ln	declarations for C Library functions (binary format; source is in **lib:llib-lc)**
lib:llib-port.ln	declarations for portable functions (binary format; source is in **lib:llib-port)**
lib:llib-lm.ln	declarations for Math Library functions (binary format; source is in **lib:llib-lm)**
/usr/tmp/*lint*	temporaries

SEE ALSO

cc(1), cpp(1), make(1).

BUGS

Exit(2) and other functions that do not return are not understood; this causes various inaccuracies.

NAME

ls – list contents of directory

SYNOPSIS

ls [–RadCxmlnogrtucpFbqisf] [names]

DESCRIPTION

For each directory *name*, *ls* lists the contents of the directory; for each file *name*, *ls* repeats its name and any other information requested. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are sorted appropriately, but file arguments appear before directories and their contents.

There are three major listing formats. The default format is to list one entry per line. The **–C** and **–x** options enable multi-column formats, and the **–m** option enables stream output format, in which files are listed across the page, separated by commas. In order to determine output formats for the **–C**, **–x**, and **–m** options, *ls* uses an environment variable, **COLUMNS**, to determine the number of character positions available on one output line. If this variable is not set, the *terminfo* database is used to determine the number of columns, based on the environment variable **TERM**. If this information cannot be obtained, *ls* assumes 80 columns.

There are a large number of options:

–R Recursively list subdirectories encountered.

–a List all entries; normally entries whose names begin with a period (.) are not listed.

–d If an argument is a directory, list only its name (not its contents); often used with **–l** to get the status of a directory.

–C Multi-column output with entries sorted down the columns.

–x Multi-column output with entries sorted across rather than down the page.

–m Stream output format.

–l List in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file (see below). If the file is a special file, the size field will instead contain the major and minor device numbers rather than a size.

–n The same as **–l**, except that the owner's **UID** and group's **GID** numbers are printed, rather than the associated character strings.

-o The same as **-l**, except that the group is not printed.

-g The same as **-l**, except that the owner is not printed.

-r Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

-t Sort by time modified (latest first) instead of by name.

-u Use time of last access instead of last modification for sorting (with the **-t** option) or printing (with the **-l** option).

-c Use time of last modification of the i-node (file created, mode changed, etc.) for sorting (**-t**) or printing (**-l**). (This is equivalent to -t for QNIX).

-p Put a slash (/) after each filename if that file is a directory.

-F Put a slash (/) after each filename if that file is a directory and put an asterisk (*) after each filename if that file is executable (Equivalent to -p on QNIX).

-b Force printing of non-graphic characters to be in the octal \ddd notation.

-q Force printing of non-graphic characters in filenames as the character (?).

-in Print the i-number of each file in the first column of the report. Note the comments on validity of i-numbers in the document "QNIX User Guide" in this manual

-s Give size in blocks, including indirect blocks, for each entry.

-f Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off **-l**, **-t**, **-s**, and **-r**, and turns on **-a**; the order is the order in which entries appear in the directory.

The mode printed under the **-l** option consists of 10 characters that are interpreted as follows:

The first character is:

d if the entry is a directory;
c if the entry is a character special file;
p if the entry is a fifo ("named pipe") special file;
- if the entry is an ordinary file.

The next 9 characters are interpreted as three sets of three bits

each. The first set refers to the owner's permissions; the next to permissions of others in the user-group of the file; and the last to all others. Within each set, the three characters indicate permission to read, write, and execute the file as a program, respectively. For a directory, "execute" permission is interpreted to mean permission to search the directory for a specified file.

The permissions are indicated as follows:

- **r** if the file is readable;
- **w** if the file is writable;
- **x** if the file is executable;
- **–** if the indicated permission is *not* granted.

QNIX does not support user groups or group file permissions. The group permissions are copied from the "other" field, and execute permission is never shown for files.

When the sizes of the files in a directory are listed, a total count of blocks, including file information blocks, is printed.

SEE ALSO

chmod(2).

BUGS

Unprintable characters in filenames may confuse the columnar output options.

QNIX does not interpret user IDs as user names at present.

NAME

make – maintain, update, and regenerate groups of programs

SYNOPSIS

make [**–f** makefile] [**–p**] [**–i**] [**–k**] [**–s**] [**–r**] [**–n**] [**–b**] [**–e**] [**–t**] [**–d**] [**–q**] [names]

DESCRIPTION

Make executes commands in *makefile* to update one or more target *names*. The *names* are typically programs.

Make updates a target only if its dependents are newer than the target. All prerequisite files of a target are added recursively to the list of targets. Missing files are considered out of date.

Makefile should contain a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, non-null list of targets, then a :, then a (possibly null) list of prerequisite files or dependencies. Text following a ; and all following lines that begin with a tab are shell commands to be executed to update the target(s). The first line that does not begin with a tab or # begins a new dependency or macro definition (see *Macros* below). Shell commands may be continued across lines with the <backslash><newline> sequence. Sharp (#) and newline surround comments.

Command Execution

Command lines are executed one at a time, each by its own shell. Commands are normally printed when they are executed. Everything printed by *make* (except the initial tab) is passed directly to the shell as is. Thus,

```
echo a\
b
```

will produce

```
ab
```

exactly the same as the shell would.

The first one or two characters on a command line can be one or both of the following: **–**, **@**, **–@**, or **@–**. If **–** is present, any error that occurs when the command is executed is ignored. If **@** is present, the command line is not printed. The command line is also not printed if the **–s** option is present, or if the entry **.SILENT:** is present in *makefile*; see *Options* below.

The **−n** option specifies printing of the command lines without execution; however, if the command line has the string **$(MAKE)** in it, the command is always executed (see discussion of the **MAKEFLAGS** macro under *Environment*).

The **−t** option updates the modified date of a file without executing any commands.

Commands that return non-zero status normally terminate *make*. If **−** appears as one of the first two characters on the line, or if the **−i** option is used, or if the entry **.IGNORE:** appears in *makefile*, the error is ignored. If the **−k** option is used, work is abandoned on the current entry, but continues on other branches that do not depend on that entry.

The **−b** option allows old makefiles (those written for the old version of *make*), to run without errors. The difference between the old and present versions of *make* is that this version requires all dependency lines to have a (possibly null or implicit) command associated with them. The previous version of *make* assumed that, if no command was given, the command was null.

Interrupt and quit cause the target to be deleted unless the target is a dependent of the special name **.PRECIOUS.**

For full descriptions of all options and special names for *make*, see the *Options* section below.

Makefile Example

The following *makefile* says that **pgm** depends on two files **a.o** and **b.o**, and that they in turn depend on their corresponding source files (**a.c** and **b.c**) and a common file **incl.h**. It also updates **pgm**, **a.o**, and **b.o** (if they are out of date) by running each one's prerequisite files through *cc*(1).

```
pgm: a.o b.o
        cc a.o b.o −o pgm
a.o: incl.h a.c
        cc −c a.c
b.o: incl.h b.c
        cc −c b.c
```

Options

The following is a brief description of all options and some special names for *make*:

−f *makefile* Description filename. *Makefile* is assumed to be the name of a description file. A filename of **−** denotes the standard

input. If *makefile* is specified, its contents override the built-in rules. If no **−f** option is present, **makefile**, **Makefile**, **s.makefile**, and **s.Makefile** are tried in order. More than one **−f** *makefile* argument pair may appear.

−p Print out the complete set of macro definitions and target descriptions.

−i Ignore error codes returned by invoked commands. This mode is also entered if the fake target name **.IGNORE** appears in the description file.

−k Abandon work on the current entry if an error occurs, but continue on other branches that do not depend on that entry.

−s Do not print command lines before executing. This mode is also entered if the fake target name **.SILENT** appears in the description file.

−r Do not use the built-in rules.

−n Print commands, but do not execute them. Even lines beginning with an **@** are printed.

−b Compatibility mode for old makefiles.

−e Override assignments within makefiles with environment variables.

−t Touch the target files (causing them to be up-to-date) without issuing any commands.

−d Debug mode. Print out detailed information on files and times examined.

−q Question. The *make* command returns a zero or non-zero status code depending on whether the target file is or is not up-to-date.

.DEFAULT If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name **.DEFAULT** are used if it exists.

.PRECIOUS Dependents of this target will not be removed when quit or interrupt are hit.

.SILENT Same effect as the **−s** option.

.IGNORE Same effect as the **−i** option.

Environment

The environment is read by *make*. All variables are treated as macro definitions. The environment variables are processed before any makefile and after the internal rules; thus, macro assignments in a makefile override environment variables. The **−e** option causes the environment to override the macro assignments in a makefile.

The **MAKEFLAGS** environment variable is processed by *make* as containing any legal input option (except **−f**, **−p**, and **−d**) usable in the command line. Upon invocation, *make* "invents" the variable if it is not in the environment, puts the current options into it, and passes it on to invocations of commands. Thus, **MAKEFLAGS** always contains the current input options. This is very useful for "super-makes." When the **−n** option is used, the command **$(MAKE)** is executed; hence, one can perform a **make −n** recursively on a whole software system to see what would have been executed. This is because the **−n** is put in **MAKEFLAGS** and passed to further invocations of **$(MAKE)**. This is one way of debugging all of the makefiles for a software project without actually doing anything.

Macros

Entries of the form *string1* = *string2* are macro definitions. *String2* is defined as all characters up to a comment character or an unescaped newline. Subsequent appearances of $(*string1*[:*subst1*=[*subst2*]]) are replaced by *string2*. The optional :*subst1*=*subst2* is a substitute sequence. The parentheses are optional if a single character macro name is used and there is no substitute sequence. If a substitute sequence is specified, all nonoverlapping occurrences of *subst1* in the named macro are replaced by *subst2*. Strings (for the purposes of this type of substitution) are delimited by blanks, tabs, newline characters, and beginnings of lines. An example of the use of the substitute sequence is shown under *Libraries*.

Internal Macros

There are five internally maintained macros that are useful for writing rules for building targets.

$*** The **$* macro stands for the filename part of the current dependent with the suffix deleted. It is evaluated only for inference rules.

$@** The **$@ macro stands for the full target name of the current target. It is evaluated only for explicitly-named dependencies.

$<** The **$< macro is only evaluated for inference rules or the **.DEFAULT** rule. It is the module that is out-of-date with respect to the target

(i.e., the "manufactured" dependent filename). Thus, in the **.c.o** rule, the **$<** macro would evaluate to the **.c** file. An example for making optimized **.o** files from **.c** files is:

```
.c.o:
        cc -c -O $*.c
```

or:

```
.c.o:
        cc -c -O $<
```

$?** The **$? macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are out of date with respect to the target; essentially, those modules that must be rebuilt.

$% The **$%** macro is only evaluated when the target is an archive library member of the form **lib(file.o)**. In this case, **$@** evaluates to **lib** and **$%** evaluates to the library member, **file.o**.

Four of the five macros can have alternative forms. When an upper case **D** or **F** is appended to any of the four macros, the meaning is changed to "directory part" for **D** and "file part" for **F**. Thus, **$(@D)** refers to the directory part of the string **$@**. If there is no directory part, **./** is generated. The only macro excluded from this alternative form is **$?**.

Suffixes and Inference Rules

Make has inference rules that allow the building of files from other files by examining the suffixes and determining an appropriate inference rule to use. Certain names (for instance, those ending with **.o**) have prerequisites (such as **.c**, **.s**, etc) that can be inferred. If no update commands for such a file appear in *makefile*, and if a prerequisite that can be inferred exists, that prerequisite is compiled to make the target. The current default inference rules are:

```
.c .c~ .sh .sh~ .c.o .c~.o .c~.c .s.o .s~.o .y.o .y~.o .l.o .l~.o
.y.c .y~.c .l.c .c.a .c~.a .s~.a .h~.h
```

The internal rules for *make* are contained in the source file **rules.c**. These rules can be locally modified. The following command prints out the rules that are compiled into *make* in a form suitable for recompilation:

```
make -fp - 2>/dev/null </dev/null
```

The only peculiarity in this output is the **(null)** string, which *printf*(3S) prints when handed a null string.

A tilde in the above rules refers to an SCCS file. Thus, the rule **.c~.o** would transform an SCCS C source file into an object file (**.o**). Because the **s.** of the SCCS files is a prefix, it is incompatible with *make*'s suffix point-of-view. The tilde is a way of changing any file reference into an SCCS file reference.

A rule with only one suffix (i.e., **.c:**) is the definition of how to build *x* from *x*.**c**. In effect, the other suffix is null. This is useful for building targets from only one source file (e.g., shell procedures, simple C programs).

Additional suffixes are given as the dependency list for **.SUFFIXES:**. Order is significant; the first possible name for which both a file and a rule exist is inferred as a prerequisite. The default list is:

```
.SUFFIXES: .o .c .y .l .s
```

Here again, the above command for printing the internal rules will display the list of suffixes implemented on the current machine. Multiple suffix lists accumulate; **.SUFFIXES:** with no dependencies clears the list of suffixes.

Because of the inference rules, the example *makefile* given above can be done more briefly:

```
pgm: a.o b.o
        cc a.o b.o -o pgm
a.o b.o: incl.h
```

The user may add rules to the list of internal rules by simply including them in the *makefile*.

Certain macros are used by the default inference rules to permit the inclusion of optional matter in any resulting commands. For example, **CFLAGS**, **LFLAGS**, and **YFLAGS** are used for compiler options to *cc*(1), *lex*(1), and *yacc*(1), respectively. Again, the previous method for examining the current rules can be used.

The inference of prerequisites can be controlled. The rule to create a file with suffix **.o** from a file with suffix **.c** is specified as an entry with **.c.o:** as the target and no dependents. Shell commands associated with the target define the rule for making a **.o** file from a **.c** file. Any target that has no slashes in it and starts with a dot is identified as a rule and not a true target.

Libraries

If a target or dependency name contains parentheses, it is assumed to be an archive library, the string within parentheses referring to a member

within the library. Thus **lib(file.o)** and **$(LIB)(file.o)** both refer to an archive library that contains **file.o** (assuming the **LIB** macro has been previously defined). The expression **$(LIB)(file1.o file2.o)** is not legal. Rules pertaining to archive libraries have the form *.XX*.**a** where the *XX* is the suffix from which the archive member is to be made. A byproduct of the current implementation requires the *XX* to be different from the suffix of the archive member. Thus, one cannot have **lib(file.o)** depend upon **file.o** explicitly. The most common use of the archive interface follows. Here, we assume the source files are all C type sources:

```
lib:    lib(file1.o) lib(file2.o) lib(file3.o)
        @echo lib is now up-to-date
.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
```

In fact, the **.c.a** rule listed above is built into *make* and is unnecessary in this example. A more interesting, but more limited example of an archive library maintenance construction follows:

```
lib:    lib(file1.o) lib(file2.o) lib(file3.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        ar rv lib $?
        rm $?
        @echo lib is now up-to-date
.c.a:;
```

Here the substitution mode of the macro expansions is used. The **$?** list is defined to be the set of object filenames (inside **lib**) whose C source files are out-of-date. The substitution mode translates the **.o** to **.c**. (Presently it is not possible to transform to **.c~**.) Note also, the disabling of the **.c.a:** rule, which would have created each object file, one by one. This particular construct speeds up archive library maintenance considerably. This type of construct becomes very cumbersome if the archive library contains a mix of assembly programs and C programs.

FILES

[Mm]akefile and s.[Mm]akefile

SEE ALSO

cc(1), cd(1), lex(1), sh(1), yacc(1).
printf(3S).
The documents "A Program for Maintaining Computer Programs (make)"

and "Augmented Version of make" in this manual.

BUGS

Some commands return non-zero status inappropriately; use **-i** to overcome this difficulty. Filenames with the characters = : @ will not work. Commands that are directly executed by the shell, notably *cd*(1), are ineffectual across newlines in *make*. The syntax **(lib(file1.o file2.o file3.o)** is illegal. You cannot build **lib(file.o)** from **file.o**. The macro **$(a:.o=.c~)** does not work.

NAME

mkdir – make a directory

SYNOPSIS

mkdir dirname ...

DESCRIPTION

Mkdir creates specified directories. The new directories have mode 777 (possibly altered by *umask*(1)). Standard entries (. for the directory itself and .. for its parent) are made automatically.

Mkdir requires write permission in the parent directory.

SEE ALSO

sh(1), rm(1).

DIAGNOSTICS

Mkdir returns exit code 0 if all directories were successfully made; otherwise, it prints a diagnostic and returns non-zero.

NAME

nl – line numbering filter

SYNOPSIS

nl [–**h**type] [–**b**type] [–**f**type] [–**v**start#] [–**i**incr] [–**p**] [–**l**num] [–**s**sep] [–**w**width] [–**n**format] [–**d**delim] file

DESCRIPTION

Nl reads lines from the named *file*, or the standard input if no *file* is named, and reproduces the lines on the standard output. Lines are numbered on the left in accordance with the command options in effect.

Nl views the text it reads in terms of logical pages. Line numbering is reset at the start of each logical page. A logical page consists of a header, a body, and a footer section. Empty sections are valid. Different line numbering options are independently available for header, body, and footer.

The start of logical page sections are signaled by input lines containing nothing but the following delimiter character(s):

Line contents:	*Start of:*
\:\:\:	header
\:\:	body
\:	footer

Unless optioned otherwise, *nl* assumes the text being read is in a single logical page body.

Command options may appear in any order and may be intermingled with the optional *file* argument. Only one file may be named. The options are:

–**b***type* Specify which lines in the logical page body are to be numbered. Recognized *types* and their meaning are:

a number all lines.
t number lines with printable text only; this is the default.
n no line numbering.
p*string*
number only lines that contain the regular expression specified in *string*.

−h*type* Same as the **−b** option, except it is for lines in the header. Default *type* for the logical page header is **n** (no lines numbered).

−f*type* Same as the **−b** option, except it is for lines in the footer. Default for logical page footer is **n** (no lines numbered).

−p Do not restart numbering at logical page delimiters.

−v*start#* Use *start#* as the initial value for numbering logical page lines. Default is **1**.

−i*incr* Use *incr* as the increment value for numbering logical page lines. Default is **1**.

−s*sep* Use *sep* as the character(s) to separate the line number from the corresponding text line. Default *sep* is a tab.

−w*width* Use *width* number of characters for the line number. Default *width* is **6**.

−n*format* Set the line numbering format to *format*. Recognized values are:

ln left justified, leading zeroes suppressed.
rn right justified, leading zeroes supressed. This is the default.
rz right justified, leading zeroes kept.

−l*num* Consider *num* blank lines as one. For example, **−l***2* causes every other adjacent blank line to be numbered (if the appropriate **−ha**, **−ba**, and/or **−fa** option is set). Default is **1**.

−d*xx* Change the delimiter characters specifying the start of a logical page from the default characters (\:) to two user-specified characters. If only one character is entered, the second character remains the default character (:). No space should appear between the **−d** and the delimiter characters. To enter a backslash, use two backslashes.

EXAMPLE

The command:

```
nl −v10 −i10 −d!+ file1
```

will number file1 starting at line number 10 with an increment of ten. The logical page delimiters are !+.

SEE ALSO
pr(1).

NAME

od – octal dump

SYNOPSIS

od [–bcdosx] [file] [[+]offset[.][b]]

DESCRIPTION

Od dumps *file* in one or more formats as selected by the first argument. If the first argument is missing, **–o** is the default. The first argument must be one of the following:

–b Interpret bytes in octal.

–c Interpret bytes in ASCII. Certain non-graphic characters appear as C escapes: null=**\0**, backspace=**\b**, form-feed=**\f**, newline=**\n**, return=**\r**, tab=**\t**; others appear as 3-digit octal numbers.

–d Interpret words in unsigned decimal.

–o Interpret words in octal.

–s Interpret 16-bit words in signed decimal.

–x Interpret words in hex.

File is the file to be dumped. If no file argument is specified, the standard input is used.

Offset specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If **.** is appended, the offset is interpreted in decimal. If **b** is appended, the offset is interpreted in blocks of 512 bytes. If the file argument is omitted, the offset argument must be preceded by **+**.

Dumping continues until end-of-file.

NAME

paste – merge same lines of several files or subsequent lines of one file

SYNOPSIS

paste file1 file2 ...
paste **–d**list file1 file2 ...
paste **–s** [**–d**list] file1 file2 ...

DESCRIPTION

In the first two forms, *paste* concatenates corresponding lines of the given input files *file1*, *file2*, etc. It treats each file as a column or columns of a table and pastes them together horizontally (parallel merging). In the third form, *paste* replaces the function of an older command with the same name by combining subsequent lines of the input file(s) (serial merging). In all cases, lines are connected with the *tab* character, or with characters from an optionally specified *list*.

Output is to the standard output. *Paste* can be used as the start of a pipe, or as a filter by specifying - (the standard input) instead of a filename.

Available options are:

–d By default, the newline characters of each but the last file (or last line in case of the **–s** option) are replaced by a *tab* character. This option replaces the *tab* character with one or more alternate characters (see *list* below).

list One or more characters immediately following **–d** replace the default *tab* as the line concatenation character. The list is used circularly; when exhausted, it is reused. In parallel merging (i.e., no **–s** option), the lines from the last file are always terminated with a newline character, not with a character from the *list*. The list may contain the following special escape sequences: **\n** (newline), **\t** (tab), **** (backslash), and **\0** (empty string, not a null character). Quoting may be necessary if characters have special meaning to the shell (for example, to get one backslash, use *–d*"\\\\").

–s Merge subsequent lines rather than one from each input file. This option uses *tab* for concatenation, unless a *list* is specified with the **–d** option. Regardless of the *list*, the very last character of the file is forced to be a newline.

– May be used in place of any filename to read a line from the standard input. (There is no prompting).

EXAMPLES

ls | paste −d" " −	list directory in one column
ls | paste − − − −	list directory in four columns
paste −s −d"\t\n" file	combine pairs of lines into lines

SEE ALSO

cut(1), grep(1), pr(1).

DIAGNOSTICS

line too long	Output lines are restricted to 511 characters.
too many files	Except for −s option, no more than 12 input files may be specified.

NAME

pr – print files

SYNOPSIS

pr [options] [files]

DESCRIPTION

Pr prints the named files on the standard output. If *file* is –, or if no files are specified, the standard input is assumed. By default, the listing is separated into pages, each headed by the page number, date and time, and the name of the file.

By default, columns are of equal width, separated by at least one space; lines that do not fit are truncated.

If the standard output is associated with a terminal, error messages are withheld until *pr* has completed printing.

The following *options* may appear singly or be combined in any order:

+*k* Begin printing with page *k* (default is 1).

–*k* Produce *k*-column output (default is 1). The **–e** and **–i** options are assumed for multi-column output.

–a Print multi-column output across the page.

–m Merge and print all files simultaneously, one per column (overrides the –*k* and **–a** options).

–d Double-space the output.

–e*ck* Expand *input* tabs to character positions *k*+1, 2**k*+1, 3**k*+1, etc. If *k* is 0 or is omitted, default tab settings at every eighth position are assumed. Tab characters in the input are expanded into the appropriate number of spaces. If *c* (any non-digit character) is given, it is treated as the input tab character (default for *c* is the tab character).

–i*ck* Replace white space in the output wherever possible by inserting tabs to character positions *k*+1, 2**k*+1, 3**k*+1, etc. If *k* is 0 or is omitted, default tab settings at every eighth position are assumed. If *c* (any non-digit character) is given, it is treated as the output tab character (default for *c* is the tab character).

–n*ck* Provide *k*-digit line numbering (default for *k* is 5). The number occupies the first *k*+1 character positions of each column of normal output or each line of **–m** output. If *c* (any non-digit character) is given, it is appended to the line number to separate it

from whatever follows (default for *c* is a tab).

−w*k* Set the width of a line to *k* character positions (default is 72 for equal-width multi-column output, no limit otherwise).

−o*k* Offset each line by *k* character positions (default is 0). The number of character positions per line is the sum of the width and offset.

−l*k* Set the length of a page to *k* lines (default is 66).

−h Use the next argument as the header to be printed instead of the filename.

−p Pause before beginning each page if the output is directed to a terminal (*pr* will ring the bell at the terminal and wait for a carriage return).

−f Use form-feed character for new pages (default is to use a sequence of line-feeds). Pause before beginning the first page if the standard output is associated with a terminal.

−r Do not print diagnostic reports on failure to open files.

−t Do not print either the five-line identifying header or the five-line trailer normally supplied for each page. Quit printing after the last line of each file without spacing to the end of the page.

−s*c* Separate columns by the single character *c* instead of by the appropriate number of spaces (default for *c* is a tab).

EXAMPLES

Print **file1** and **file2** as a double-spaced, three-column listing headed by "file list":

```
pr −3dh "file list" file1 file2
```

Write **file1** on **file2**, expanding tabs to columns 10, 19, 28, 37, ... :

```
pr −e9 −t <file1 >file2
```

FILES

/dev/tty* to suspend messages

SEE ALSO

cat(1).

NAME

pwd – working directory name

SYNOPSIS

pwd

DESCRIPTION

Pwd prints the pathname of the working (current) directory.

BUGS

Pwd may expand the pathname of the working directory fully if the first component of the path is a wellknown prefix.

SEE ALSO

cd(1).

NAME

rm, rmdir – remove files or directories

SYNOPSIS

rm [**–fri**] file ...

rmdir dir ...

DESCRIPTION

Rm removes the entries for one or more files from a directory. If an entry was the last link to the file, the file is destroyed. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself.

If a file has no write permission and the standard input is a terminal, its permissions are printed and a line is read from the standard input. If that line begins with **y** the file is deleted, otherwise the file remains. No questions are asked when the **–f** option is given or if the standard input is not a terminal.

If a file given to *rm* is a directory, an error comment is printed unless the optional argument **–r** has been used. In that case, *rm* recursively deletes the entire contents of the specified directory and the directory itself.

If the **–i** (interactive) option is in effect, *rm* asks whether to delete each file, and, under **–r**, whether to examine each directory.

Rmdir removes entries for the named directories, which must be empty.

SEE ALSO

unlink(2).

DIAGNOSTICS

Generally self-explanatory. It is forbidden to remove the file .. to avoid the consequences of inadvertently doing something like:

rm –r .*

NAME

sh, rsh – shell, the standard/restricted command programming language

SYNOPSIS

sh [**–acefhiknrstuvx**] [args]
rsh [**–acefhiknrstuvx**] [args]

DESCRIPTION

Sh is a command programming language that executes commands read from a terminal or a file. *Rsh* is a restricted version of the standard command interpreter *sh*; it is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. See *Invocation* below for the meaning of arguments to the shell.

Definitions

A *blank* is a tab or a space. A *name* is a sequence of letters, digits, or underscores beginning with a letter or underscore. A *parameter* is a name, a digit, or any of the characters *, @, #, ?, –, $, and !.

Commands

A *simple-command* is a sequence of non-blank *words* separated by *blanks*. The first word specifies the name of the command to be executed. Except as specified below, the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 (see *exec*(2)). The *value* of a simple-command is its exit status if it terminates normally, or (octal) 200+*status* if it terminates abnormally (see *signal*(2) for a list of status values).

A *pipeline* is a sequence of one or more *commands* separated by **|** (or, for historical compatibility, by ^). The standard output of each command but the last is connected by a *pipe*(2) to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate. The exit status of a pipeline is the exit status of the last command.

A *list* is a sequence of one or more pipelines separated by **;**, **&**, **&&**, or **||**, and optionally terminated by **;** or **&**. Of these symbols, **;** and **&** have equal precedence, and **&&** and **||** have equal precedence; **;** and **&** have lower precedence than **&&** and **||**. A semicolon (**;**) causes sequential execution of the preceding pipeline; an ampersand (**&**) causes asynchronous execution of the preceding pipeline (i.e., the shell does *not* wait for that pipeline to finish). The symbols **&&** and **||** cause the *list* following them to be executed only if the preceding pipeline returns a zero or non-zero

exit status, respectively. An arbitrary number of newlines may be used to delimit commands in a *list*, instead of using semicolons.

A *command* is either a simple-command or one of the following. Unless otherwise stated, the value returned by a command is that of the last simple-command executed in the command.

for *name* [**in** *word* ...] **do** *list* **done**
The **for** command executes the commands in the **do** list with *name* set to each *word* in the *word* list in turn, until there are no more *words* in the list. If **in** *word* ... is omitted, then the **for** command executes the **do** *list* once for each positional parameter that is set (see *Parameter Substitution* below).

case *word* **in** [*pattern* [| *pattern*] ...) *list* ;;] ... **esac**
A **case** command executes the *list* associated with the first *pattern* that matches *word*. The form of the patterns is the same as that used for file-name generation (see *Filename Generation* below) except that a slash, a leading dot, or a dot immediately following a slash need not be matched explicitly.

if *list* **then** *list* [**elif** *list* **then** *list*] ... [**else** *list*] **fi**
The *list* following **if** is executed and, if it returns a zero exit status, the *list* following the first **then** is executed. Otherwise, the *list* following **elif** is executed and, if its value is zero, the *list* following the next **then** is executed. Failing that, the **else** *list* is executed. If no **else** *list* or **then** *list* is executed, the **if** command returns a zero exit status.

while *list* **do** *list* **done**
A **while** command repeatedly executes the first *list*. If the exit status of the last command in the list is zero, the *list* following the **do** is executed; otherwise, the loop terminates. If no commands in the **do** *list* are executed, the **while** command returns a zero exit status. **Until** may be used in place of **while** to negate the loop termination test.

(*list***)**
Execute *list* in a sub-shell.

{*list***;}**
List is simply executed.

name **() {***list***;}**
Define a function that is referenced by *name*. The function's body is the *list* of commands between { and }. Execution of functions is described below (see *Execution*).

The following words are only recognized as the first word of a command and when not quoted:

if then else elif fi case esac for while until do done { }

Comments

A word beginning with # causes that word and all the following characters up to a newline to be ignored.

Command Substitution

The standard output from a command enclosed in a pair of grave accents (` `) may be used as part or all of a word; trailing newlines are removed.

Parameter Substitution

The character **$** is used to introduce substitutable *parameters*. There are two types of parameters, positional and keyword. If *parameter* is a digit, it is a positional parameter. Positional parameters may be assigned values by **set**. Keyword parameters (also known as variables) may be assigned values by writing:

name=*value* [*name*=*value*]...

Pattern-matching is not performed on *value*. There cannot be a function and a variable with the same *name*.

${*parameter***}**

The value, if any, of the parameter is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If *parameter* is * or @, all the positional parameters, starting with **$1**, are substituted (separated by spaces); see *Quoting* below. Parameter **$0** is set from argument zero when the shell is invoked.

${*parameter***:-***word***}**

If *parameter* is set and is non-null, substitute its value; otherwise substitute *word*.

${*parameter***:=***word***}**

If *parameter* is not set or is null, set it to *word*; otherwise the value of the parameter is substituted. Positional parameters may not be assigned in this way.

${*parameter***:?***word***}**

If *parameter* is set and is non-null, substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted, the message "parameter null or not set" is printed.

${*parameter***:+***word***}**

If *parameter* is set and is non-null, substitute *word*; otherwise

substitute nothing.

In the above, *word* is not evaluated unless it is to be used as the substituted string. In the following example, **pwd** is executed only if **d** is not set or is null:

```
echo ${d:- `pwd `}
```

If the colon (:) is omitted from the above expressions, the shell only check whether *parameter* is set or not.

The following parameters are automatically set by the shell:

- **#** The number of positional parameters in decimal.
- **-** Flags supplied to the shell on invocation or by the **set** command.
- **?** The decimal value returned by the last synchronously executed command.
- **$** The process number of this shell.
- **!** The process number of the last background command invoked.

The following parameters are used by the shell:

- **HOME** The default argument (home directory) for the *cd* command.
- **PATH** The search path for commands (see *Execution* below). Since the **PATH** list is separated by colons, environment searchlists may not be used.
- **CDPATH** The search path for the *cd* command.
- **PS1** Primary prompt string, by default "**$** ".
- **PS2** Secondary prompt string, by default "> ".
- **IFS** Internal field separators, normally **space**, **tab**, and **newline**.
- **SHELL** When the shell is invoked, it scans the environment (see *Environment* below) for this name. If it is found and there is an 'r' in the filename part of its value, the shell becomes a restricted shell.

The shell gives default values to **PATH**, **PS1**, **PS2**, and **IFS**.

Blank Interpretation

After parameter and command substitution, the results of substitution are scanned for internal field separator characters (those found in **IFS**) and split into distinct arguments where such characters are found. Explicit null arguments ("" or '') are retained. Implicit null arguments (those

resulting from *parameters* that have no values) are removed.

Filename Generation

Following substitution, each command *word* is scanned for the characters *, ?, and [. If one of these characters appears, the word is regarded as a *pattern*. The word is replaced with alphabetically sorted filenames that match the pattern. If no filename is found that matches the pattern, the word is left unchanged. The character . at the start of a filename or immediately following a /, as well as the character / itself, must be matched explicitly.

*	Matches any string, including the null string.
?	Matches any single character.
[...]	Matches any one of the enclosed characters. A pair of characters separated by – matches any character lexically between the pair, inclusive. If the first character following the opening [is a !, then any character not enclosed is matched.

Quoting

The following characters have a special meaning to the shell and cause termination of a word unless quoted:

; & ( ) | ^ < > newline space tab

A character may be *quoted* (i.e., made to stand for itself) by preceding it with a \. The pair **\newline** is ignored. All characters enclosed between a pair of single quote marks (' '), except a single quote, are quoted. Inside double quote marks (" "), parameter and command substitution occurs and the characters \, `, ", and $ must be quoted with \. **"$*"** is equivalent to **"$1 $2 ..."**, and **"$@"** is equivalent to **"$1" "$2" ....**

Prompting

When used interactively, the shell prompts with the value of **PS1** before reading a command. If at any time a newline is typed and further input is needed to complete a command, the secondary prompt (i.e., the value of **PS2**) is issued.

Input/Output

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a simple-command or may precede or follow a *command* and are *not* passed on to the invoked command; substitution occurs before *word* or *digit* is used:

<word	Use file *word* as standard input (file descriptor 0).
>word	Use file *word* as standard output (file descriptor 1). If the file does not exist it is created; otherwise, it is truncated to zero length.
>>word	Use file *word* as standard output. If the file exists output is appended to it (by first seeking to the end-of-file); otherwise, the file is created.
<<[–]word	The shell input is read up to a line that is the same as *word*, or to an end-of-file. The resulting document becomes the standard input. If any character of *word* is quoted, the characters of the document are not interpreted; otherwise, parameter and command substitution occurs, (unescaped) **\newline** is ignored, and \ must be used to quote the characters \, **$**, `, and the first character of *word*. If **–** is appended to **<<**, all leading tabs are stripped from *word* and from the document.
<&digit	Use the file associated with file descriptor *digit* as standard input. Similarly for the standard output using **>&digit**.
<&–	The standard input is closed. Similarly for the standard output using **>&–**.

If any of the above is preceded by a digit, the digit specifies the file descriptor that will be associated with the file (instead of the default 0 or 1). For example:

... 2>&1

associates file descriptor 2 with the file currently associated with file descriptor 1.

The order in which redirections are specified is significant. The shell evaluates redirections left-to-right. For example:

... 1>*xxx* 2>&1

first associates file descriptor 1 with file *xxx*. It associates file descriptor 2 with the file associated with file descriptor 1 (i.e. *xxx*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been) and file descriptor 1 would be associated with file *xxx*.

If a command is followed by **&** the default standard input for the command is the empty file **/dev/null**. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input/output specifications.

Redirection of output is not allowed in the restricted shell.

Environment

The *environment* (see *environ*(5)) is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list. The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a parameter for each name found, giving it the corresponding value. If the user modifies the value of any of these parameters or creates new parameters, none of these affects the environment unless the **export** command is used to bind the shell's parameter to the environment (see also **set −a**). A parameter may be removed from the environment with the **unset** command. The environment seen by an executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, minus any pairs removed by **unset**, plus any modifications or additions, all of which must be noted in **export** commands.

The environment for any *simple-command* may be augmented by prefixing it with one or more assignments to parameters. Thus:

```
TERM=450 cmd                    and
(export TERM; TERM=450; cmd)
```

are equivalent (as far as the execution of *cmd* is concerned).

If the **−k** flag is set, *all* keyword arguments are placed in the environment, even if they occur after the command name. The following first prints **a=b c** and then **c**:

```
echo a=b c
set −k
echo a=b c
```

Signals

The **INTERRUPT** and **QUIT** signals for an invoked command are ignored if the command is followed by **&**; otherwise signals have the values inherited by the shell from its parent, with the exception of signal 11 (see also the **trap** command below).

Execution

Each time a command is executed, the above substitutions are carried out. If the command name matches one of the *Special Commands* listed below, it is executed in the shell process. If the command name does not match a *Special Command*, but matches the name of a defined function, the function is executed in the shell process (note how this differs from the

execution of shell procedures). The positional parameters **$1**, **$2**, ... are set to the arguments of the function. If the command name matches neither a *Special Command* nor the name of a defined function, a new process is created and an attempt is made to execute the command via *exec*(2).

The shell parameter **PATH** defines the search path for the directory containing the command. Alternative directory names are separated by a colon (:). The default path is **:/bin:/usr/bin** (specifying the current directory, **/bin**, and **/usr/bin**, in that order). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If the command name contains a **/**, the search path is not used; such commands will not be executed by the restricted shell. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an **a.out** file, it is assumed to be a file containing shell commands, and a sub-shell is spawned to read it. A parenthesized command is also executed in a sub-shell.

The location in the search path where a command was found is remembered by the shell (to help avoid unnecessary *execs* later). If the command was found in a relative directory, its location must be re-determined whenever the current directory changes. The shell forgets all remembered locations whenever the **PATH** variable is changed or the **hash -r** command is executed (see below).

Special Commands

Input/output redirection is permitted for the following commands. File descriptor 1 is the default output location.

: No effect; the command does nothing. A zero exit code is returned.

. *file* Read and execute commands from *file* and return. The search path specified by **PATH** is used to find the directory containing *file*.

break [*n*]
Exit from the enclosing **for** or **while** loop, if any. If *n* is specified, break *n* levels.

continue [*n*]
Resume the next iteration of the enclosing **for** or **while** loop. If *n* is specified resume at the *n*-th enclosing loop.

cd [*arg*]
Change the current directory to *arg*. The shell parameter **HOME** is the default *arg*. The shell parameter **CDPATH** defines the search path for the directory containing *arg*. Alternative directory names are separated by a colon (:). The default path is **<null>** (specifying the current directory). Note that the current directory is specified by a null pathname, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If *arg* begins with a **/**, the search path is not used. Otherwise, each directory in the path is searched for *arg*. The *cd* command may not be executed by *rsh*.

echo [*arg* ...]
Echo arguments. See *echo*(1) for usage and description.

eval [*arg* ...]
Read the *args* as input to the shell and execute the resulting command(s).

exec [*arg* ...]
Execute the command specified by the *args* in place of this shell without creating a new process. Input/output arguments may appear and, if no other arguments are given, cause the shell input/output to be modified.

exit [*n*]
Exit a shell with the exit status specified by *n*. If *n* is omitted the exit status is that of the last command executed (an end-of-file also causes the shell to exit.)

export [*name* ...]
Mark the given *names* for automatic export to the *environment* of subsequently-executed commands. If no arguments are given, a list of all names that are exported in this shell is printed. Function names may *not* be exported. Exported names are registered with the Accent Environment Manager prior to the execution of commands.

hash [**−r**] [*name* ...]
Determine and remember the location in the search path of the command specified by each *name*. The **-r** option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is presented. *Hits* is the number of times a command has been invoked by the shell process. *Cost* is a measure of the work required to locate a command in the search path. There are certain situations that require the stored location of a command to be recalculated. Commands for which this will be done are indicated by an asterisk (*) adjacent to the *hits* information. *Cost* will be incremented when the recalculation is done.

pwd Print the current working directory, which is stored in the Accent environment variable **current:**. See *pwd*(1) for usage and description.

read [*name* ...]
Read a line from the standard input and assign the first word to the first *name*, the second word to the second *name*, etc., with leftover words assigned to the last *name*. The return code is 0 unless an end-of-file is encountered.

readonly [*name* ...]
Mark the given *names readonly*. The values of these *names* may not be changed by subsequent assignment. If no arguments are given, a list of all *readonly* names is printed.

return [*n*]
Exit a function with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed.

set [**−−acefhkntuvx** [*arg* ...]]

−a Mark variables that are modified or created for export.

−e Exit immediately if a command exits with a non-zero exit status.

−f Disable filename generation.

−h Locate and remember function commands as functions are defined (function commands are normally located when the function is executed).

−k Place all keyword arguments in the environment for a command, not just those that precede the command name.

−n Read commands but do not execute them.

−t Exit after reading and executing one command.

−u Treat unset variables as an error when substituting.

−v Print shell input lines as they are read.

−x Print commands and their arguments as they are executed.

−− Do not change any of the flags; useful in setting **$1** to **−**.

Using **+** rather than **−** turns off the corresponding flag. These flags can also be used upon invocation of the shell. The current set of flags is in **$−**. The remaining arguments to **set** are positional parameters and are assigned, in order, to **$1**, **$2**, ... If no arguments are given the values of all names are printed.

shift [*n*]

Rename the positional parameters from **$n+1** ... to **$1** .... If *n* is not given, it is assumed to be 1.

test Evaluate conditional expressions. See *test*(1) for usage and description.

times Print the accumulated user and system times for processes run from the shell.

trap [*arg*] [*n*] ...

Read and execute the command *arg* when the shell receives signal(s) *n*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Trap commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. An attempt to trap on signal 11 (memory fault) produces an error. If *arg* is absent all trap(s) for *n* are reset to their original values. If *arg* is the null string, this signal is ignored by the shell and by the commands it invokes. If *n* is 0 the command *arg* is executed on

exit from the shell. The **trap** command with no arguments prints a list of commands associated with each signal number.

type [*name* ...]
Indicate how *name* would be interpreted if it was used as a command name.

umask [*nnn*]
Set the user file-creation mask to *nnn* (see *umask*(2)). If *nnn* is omitted, the current value of the mask is printed.

unset [*name* ...]
For each *name*, remove the corresponding variable or function. The variables **PATH, PS1, PS2,** and **IFS** cannot be unset.

wait [*n*]
Wait for the specified process and report its termination status. If *n* is not given all currently active child processes are waited for and the return code is zero.

Invocation

If the shell is invoked through *exec*(2) and the first character of argument zero is −, commands are initially read from **/etc/profile** and from **$HOME/.profile**, if such files exist. Thereafter, commands are read as described below, This is also how the shell works when invoked as **/bin/sh**. The flags below are interpreted by the shell on invocation only. Note that unless the **−c** or **−s** flag is specified, the first argument is assumed to be the name of a file containing commands, and the remaining arguments are passed as positional parameters to that command file.

−c *string* If the **−c** flag is present, commands are read from *string*.

−s If the **−s** flag is present or if no arguments remain, commands are read from the standard input. Any remaining arguments specify positional parameters. Shell output (except for *Special Commands*) is written to file descriptor 2.

−i If the **−i** flag is present or if the shell input and output are attached to a terminal, this shell is *interactive*. In this case **TERMINATE** is ignored (so that **kill 0** does not kill an interactive shell) and **INTERRUPT** is caught and ignored (so that **wait** is interruptible). In all cases, **QUIT** is ignored by the shell.

−r If the **−r** flag is present, the shell is a restricted shell.

The remaining flags and arguments are described under the **set** command above.

Rsh Only

Rsh is used to set up login names and execution environments whose

capabilities are more controlled than those of the standard shell. The actions of *rsh* are identical to those of *sh*, except that the following are disallowed:

changing directory (see *cd*(1));
setting the value of **$PATH**;
specifying path or command names containing **/**; and
redirecting output (**>** and **>>**).

The restrictions above are enforced after **.profile** is interpreted.

When a command to be executed is found to be a shell procedure, *rsh* invokes *sh* to execute it. Thus, it is possible to provide to the end-user shell procedures that have access to the full power of the standard shell, while imposing a limited menu of commands; this scheme assumes that the end-user does not have write and execute permissions in the same directory.

The net effect of these rules is that the writer of the **.profile** has complete control over user actions, by performing guaranteed setup actions and leaving the user in an appropriate directory (probably *not* the login directory).

The system administrator often sets up a directory of commands (i.e., **/usr/rbin**) that can be safely invoked by *rsh*. QNIX also provides a restricted editor *red* (see *ed*(1)).

EXIT STATUS

Errors detected by the shell, such as syntax errors, cause the shell to return a non-zero exit status. If the shell is being used non-interactively, execution of the shell file is abandoned. Otherwise, the shell returns the exit status of the last command executed (see also the **exit** command above).

FILES

/etc/profile
$HOME/.profile
/tmp/sh*
/dev/null

SEE ALSO

cd(1), echo(1), pwd(1), test(1).
dup(2), exec(2), fork(2), pipe(2), signal(2), umask(2), wait(2).
environ(5).

CAVEATS

If a command is executed, and a command with the same name is installed in a directory in the search path before the directory where the original command was found, the shell will continue to *exec* the original command. Use the **hash** command to correct this situation.

If you move the current directory or one above it, **pwd** may not give the correct response. Use the **cd** command with a full path name to correct this situation.

NAME

sort – sort and/or merge files

SYNOPSIS

sort [**–cmu**] [**–o***output*] [**–y***kmem*] [**–z***recsz*] [**–dfiMnr**] [**–btx**] [**+***pos1* [**–***pos2*]] [*files*]

DESCRIPTION

Sort sorts lines of all the named *files* together and writes the result on the standard output. The standard input is read if – is used as a filename or if no input files are named.

Comparisons are based on one or more sort keys extracted from each line of input. By default, there is one sort key, the entire input line, and ordering is lexicographic by bytes in machine collating sequence.

The following options alter the default behavior:

–c Check that the input file is already sorted according to the ordering rules; give no output unless the file is out of sort.

–m Merge only; the input files are already sorted.

–u Unique: suppress all but one in each set of lines having equal keys.

–o*output*

The argument given is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs. There may be optional blanks between **–o** and *output*.

–y*kmem*

The amount of main memory used by the sort has a large impact on its performance. Sorting a small file in a large amount of memory is wasteful. If this option is omitted, *sort* begins using a system default memory size, and continues to use more space as needed. If this option is presented with a value, *kmem*, *sort* will use that number of kilobytes of memory. If *kmem* violates the administrative minimum or maximum, the corresponding extreme value is used; thus, **–y0** is guaranteed to start with minimum memory. By convention, **–y** with no argument starts with maximum memory.

–z*recsz*

Normally, the size of the longest line read is recorded in the sort phase so that buffers can be allocated during the merge phase. If the sort phase is omitted via the **–c** or **–m** options, a system

default size will be used. Lines longer than the default size will cause *sort* to terminate abnormally. The actual number of bytes in the longest line to be merged (or some larger value) can be supplied with the *−zrecxz* option to prevent this problem.

The following options override the default ordering rules.

−d "Dictionary" order: only letters, digits and blanks (spaces and tabs) are significant in comparisons.

−f Fold lower case letters into upper case.

−i Ignore characters outside the ASCII range 040-0176 in non-numeric comparisons.

−M Compare as months. The first three non-blank characters of the field are folded to upper case and compared so that "JAN" < "FEB" < ... < "DEC". Invalid fields compare low to "JAN". The **−M** option implies the **−b** option (see below).

−n An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. The **−n** option implies the **−b** option (see below). Note that the **−b** option is only effective when restricted sort key specifications are in effect.

−r Reverse the sense of comparisons.

When ordering options appear before restricted sort key specifications, the requested ordering rules are applied globally to all sort keys. When attached to a specific sort key (described below), the specified ordering options override all global ordering options for that key.

Sort uses the notion of fields. A field is a minimal sequence of characters followed by a field separator or a newline. By default, the first blank (space or tab) of a sequence of blanks acts as the field separator. All blanks in a sequence of blanks are considered to be part of the next field; for example, all blanks at the beginning of a line are considered to be part of the first field.

The notation *+pos1 −pos2* restricts a sort key to one beginning at *pos1* and ending at *pos2*. The characters at positions *pos1* and *pos2* are included in the sort key (provided that *pos2* does not precede *pos1*). A missing *−pos2* means the end of the line.

Pos1 and *pos2* each have the form *m.n* optionally followed by one or more of the flags **bdfinr**. A starting position specified by *+m.n* is

interpreted to mean the $n+1$st character in the $m+1$st field. A missing *.n* means .0, indicating the first character of the $m+1$st field. If the **b** flag (see below) is in effect, *n* is counted from the first non-blank in the $m+1$st field; $+m.0$**b** refers to the first non-blank character in the $m+1$st field.

A last position specified by $-m.n$ is interpreted to mean the *n*th character (including separators) after the last character of the *m th* field (i.e., in the $m+1$st field). A missing *.n* means .0, indicating the last character of the *m*th field. If the **b** flag (see below) is in effect, *n* is counted from the last leading blank in the $m+1$st field; $-m.1$**b** refers to the first non-blank in the $m+1$st field.

The treatment of field separators can be altered using the options:

−t*x* Use *x* as the field separator character; *x* is not considered to be part of a field (although it may be included in a sort key). Each occurrence of *x* is significant (e.g., *xx* delimits an empty field).

−b Ignore leading blanks when determining the starting and ending positions of a restricted sort key. If the **−b** option is specified before the first **+***pos1* argument, it will be applied to all **+***pos1* arguments. Otherwise, the **b** flag may be attached independently to each **+***pos1* or **−***pos2* argument.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

EXAMPLES

Sort the contents of *infile* with the second field as the sort key:

```
sort +1 −2 infile
```

Sort, in reverse order, the contents of *infile1* and *infile2*, placing the output in *outfile* and using the first character of the second field as the sort key:

```
sort −r −o outfile +1.0 −1.2 infile1 infile2
```

Sort, in reverse order, the contents of *infile1* and *infile2* using the first non-blank character of the second field as the sort key:

```
sort −r +1.0b −1.1b infile1 infile2
```

Print the password file sorted by the numeric user ID (the third colon-separated field):

```
sort −t: +2n −3 /etc/passwd
```

Print the lines of the already sorted file *infile*, suppressing all but the first occurrence of lines having the same third field (the options **−um** with just one input file make the choice of a unique representative from a set of equal lines predictable):

sort −um +2 −3 infile

FILES

/usr/tmp/stm???

DIAGNOSTICS

Sort prints comments and exits with non-zero status for various trouble conditions (e.g., when input lines are too long) and for disorder discovered under the **−c** option.

When the last line of an input file is missing a **newline** character, *sort* appends one, prints a warning message, and continues.

NAME

tail – deliver the last part of a file

SYNOPSIS

tail [±[number][**lbc**[**f**]]] [file]

DESCRIPTION

Tail copies the named *file* to the standard output beginning at a designated place. If no file is named, the standard input is used.

Copying begins at distance +*number* from the beginning or –*number* from the end of the input (if *number* is null, the value 10 is assumed). *Number* is counted in units of lines, blocks, or characters, according to the appended option **l**, **b**, or **c**. When no units are specified, counting is by lines.

If the **–f** option is specified, and the input file is not a pipe, *tail* will not terminate after copying the rest of the file. Instead, *tail* sleeps for a short time and then tries to read and copy more records from the input file, and repeats this process until it is killed. Thus, *tail* may be used to monitor the growth of a file that is being written by some other process. For example, the command:

tail –f fred

will print the last ten lines of the file **fred**, followed by any lines that are appended to **fred** between the time *tail* is initiated and killed. As another example, the command:

tail –15cf fred

will print the last 15 characters of the file **fred**, followed by any lines that are appended to **fred** between the time *tail* is initiated and killed.

BUGS

Tails relative to the end of the file are stored in a buffer, and thus are limited in length. Anomalous behavior may happen with character special files.

NAME

tee – pipe fitting

SYNOPSIS

tee [–i] [–a] [file **] ...**

DESCRIPTION

Tee transcribes the standard input to the standard output and makes copies in the *file*s. The **–i** option ignores interrupts; the **–a** option causes the output to be appended to the *file*s rather than overwriting them.

NAME

test – condition evaluation command

SYNOPSIS

test expr
[expr]

DESCRIPTION

Test evaluates the expression *expr* and, if its value is true, returns a zero (true) exit status; otherwise, a non-zero (false) exit status is returned. *Test* also returns a non-zero exit status if there are no arguments.

The following primitives are used to construct *expr*:

–r *file*	true if *file* exists and is readable.
–w *file*	true if *file* exists and is writable.
–x *file*	true if *file* exists and is executable.
–f *file*	true if *file* exists and is a regular file.
–d *file*	true if *file* exists and is a directory.
–c *file*	true if *file* exists and is a character special file.
–b *file*	true if *file* exists and is a block special file.
–p *file*	true if *file* exists and is a named pipe (fifo).
–u *file*	true if *file* exists and its set-user-ID bit is set.
–g *file*	true if *file* exists and its set-group-ID bit is set.
–k *file*	true if *file* exists and its sticky bit is set.
–s *file*	true if *file* exists and has a size greater than zero.
–t [*fildes*]	true if the open file whose file descriptor number is *fildes* (1 by default) is associated with a terminal device.
–z *s1*	true if the length of string *s1* is zero.
–n *s1*	true if the length of the string *s1* is non-zero.
s1 = *s2*	true if strings *s1* and *s2* are identical.
s1 != *s2*	true if strings *s1* and *s2* are *not* identical.
s1	true if *s1* is *not* the null string.
n1 **–eq** *n2*	true if the integers *n1* and *n2* are algebraically equal. Any of the comparisons **–ne**, **–gt**, **–ge**, **–lt**, and **–le** may be used in place of **–eq**.

These primitives may be combined with the following operators:

!	unary negation operator.
-a	binary *and* operator.
-o	binary *or* operator (**-a** has higher precedence than **-o**).
(expr)	parentheses for grouping.

Note that all the operators and flags are separate arguments to *test*. Note also that parentheses are meaningful to the shell and, therefore, must be escaped.

SEE ALSO

sh(1).

WARNING

In the second form of the command (the one that uses **[]**, rather than the word *test*), the square brackets must be delimited by blanks.
QNIX recognizes the second form of the command, but some UNIX systems do not.

NAME

tr – translate characters

SYNOPSIS

tr [**–cds**] [string1 [string2]]

DESCRIPTION

Tr copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in *string1* are mapped into the corresponding characters of *string2*.

Any combination of the following options may be used:

–c Complement the set of characters in *string1* with respect to the universe of characters whose ASCII codes are 001 through 377 octal.

–d Delete all input characters in *string1*.

–s Squeeze all strings of repeated output characters that are in *string2* to single characters.

The following abbreviation conventions may be used to introduce ranges of characters or repeated characters into the strings:

[a–z] Stands for the string of characters whose ASCII codes run from character **a** to character **z**, inclusive.

[a*n] Stands for *n* repetitions of **a**. If the first digit of *n* is **0**, *n* is considered octal; otherwise, *n* is taken to be decimal. A zero or missing *n* is taken to be huge; this facility is useful for padding *string2*.

Any character with special meaning used in a string should be quoted with \. In addition, \ followed by 1, 2, or 3 octal digits stands for the character whose ASCII code is given by those digits.

The following example creates a list of all the words in *file1* one per line in *file2*, where a word is taken to be a maximal string of alphabetics. The strings are quoted to protect the special characters from interpretation by the shell; 012 is the ASCII code for newline.

```
tr –cs "[A–Z][a–z]" "[\012*]" <file1 >file2
```

SEE ALSO

ed(1), sh(1).
ascii(5).

BUGS

Will not handle ASCII NULL in *string1* or *string2*; always deletes NULL from input.

NAME

true, false – provide truth values

SYNOPSIS

true

false

DESCRIPTION

True does nothing, successfully. *False* does nothing, unsuccessfully. They are typically used in input to *sh*(1) such as:

```
while true
do
        command
done
```

SEE ALSO

sh(1).

DIAGNOSTICS

True has exit status zero, *false* nonzero.

NAME

wait – await completion of process

SYNOPSIS

wait

DESCRIPTION

Wait waits until all processes started with **&** have completed, and reports on abnormal terminations.

Because the *wait*(2) system call must be executed in the parent process, the shell itself executes *wait* without creating a new process.

SEE ALSO

sh(1).
wait(2).

BUGS

Not all the processes of a 3- or more-stage pipeline are children of the shell, and thus cannot be waited for.

NAME

wc – word count

SYNOPSIS

wc [–lwc] [names]

DESCRIPTION

Wc counts lines, words, and characters in the named files, or in the standard input if no *names* appear. It also keeps a total count for all named files. A word is a maximal string of characters delimited by spaces, tabs, or newlines.

The options **l**, **w**, and **c** may be used in any combination to specify that a count of lines, words, and characters are to be reported. The default is –lwc (report all three).

When *names* are specified on the command line, they will be printed along with the counts.

NAME

yacc – yet another compiler-compiler

SYNOPSIS

yacc [**–vdlt**] grammar

DESCRIPTION

Yacc converts a context-free grammar into a set of tables for a simple automaton that executes an LR(1) parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output file **y.tab.c** must be compiled by the C compiler to produce a program *yyparse*. This program must be loaded with the lexical analyzer program, *yylex*, as well as *main* and *yyerror*, an error handling routine. These routines must be supplied by the user. *Lex*(1) is useful for creating lexical analyzers usable by *yacc*.

If the **–v** flag is used, the file **y.output** is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

If the **–d** flag is used, the file **y.tab.h** is generated with the **#define** statements that associate the *yacc*-assigned "token codes" with the user-declared "token names". This allows source files other than **y.tab.c** to access the token codes.

If the **–l** flag is used, the code produced in **y.tab.c** will *not* contain any **#line** constructs. This should only be used after the grammar and the associated actions are fully debugged.

Runtime debugging code is always generated in **y.tab.c** under conditional compilation control. By default, this code is not included when **y.tab.c** is compiled. However, when *yacc*'s **–t** option is used, this debugging code will be compiled by default. Independent of whether the **–t** option was used, the runtime debugging code is under the control of **YYDEBUG**, a pre-processor symbol. If **YYDEBUG** has a non-zero value, then the debugging code is included. If its value is zero, then the code will not be included. A program produced without the runtime debugging code will be smaller and slightly faster than one with the code.

FILES

y.output	
y.tab.c	
y.tab.h	defines for token names
yacc.tmp,	temporary files
yacc.debug,	
yacc.acts	
/usr/lib/yaccpar	parser prototype for C programs

SEE ALSO

lex(1).
malloc(3X).
The document "Yet Another Compiler Compiler (yacc)" in this manual.

DIAGNOSTICS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard error output; a more detailed report is found in the **y.output** file. Similarly, if some rules are not reachable from the start symbol, this is also reported.

BUGS

Because filenames are fixed, at most one *yacc* process can be active in a given directory at a time.

PERMUTED INDEX

echo: echo arguments	.	echo(1)
ascii: map of	ASCII character set.	ascii(5)
as: PERQ	C Code assembler.	as(1)
cpp: the	C language preprocessor.	cpp(1)
ld: link editor for PERQ	C object files.	ld(1)
cb:	C program beautifier.	cb(1)
lint:	C program checker.	lint(1)
as: PERQ C	Code assembler.	as(1)
group and parent process	IDs. /get process, process	getpid(2)
as:	PERQ C Code assembler.	as(1)
ld: link editor for	PERQ C object files.	ld(1)
uname: get name of current	UNIX system.	uname(2)
of a file.	access: determine accessibility	access(2)
access: determine	accessibility of a file.	access(2)
checkpause: control signal	actions. setpause,	setpause(2)
sbrk: change data segment space	allocation. brk,	brk(2)
realloc, calloc: main memory	allocator. malloc, free,	malloc(3c)
mallinfo: fast main memory	allocator. /calloc, mallopt,	malloc(3x)
sort: sort	and/or merge files.	sort(1)
introduction to commands and	application programs. intro:	intro(1)
formatted output of a varargs	argument list. /vsprintf: print	vprintf(3s)
formatted output of a varargs	argument list. /vsprintf: print	vprintf(3x)
varargs: handle variable	argument lists.	varargs(5)
echo: echo	arguments .	echo(1)
expr: evaluate	arguments as an expression.	expr(1)
	as: PERQ C Code assembler.	as(1)
set.	ascii: map of ASCII character	ascii(5)
as: PERQ C Code	assembler.	as(1)
setbuf, setvbuf:	assign buffering to a stream.	setbuf(3s)
strtol, atol,	atoi: convert string to integer.	strtol(3c)
integer. strtol,	atol, atoi: convert string to	strtol(3c)
wait:	await completion of process.	wait(1)
processing language.	awk: pattern scanning and	awk(1)
portions of pathnames.	basename, dirname: deliver	basename(1)
cb: C program	beautifier.	cb(1)
fread, fwrite:	binary input/output.	fread(3s)
space allocation.	brk, sbrk: change data segment	brk(2)
stdio: standard	buffered input/output package.	stdio(3s)
setbuf, setvbuf: assign	buffering to a stream.	setbuf(3s)
swab: swap	bytes.	swab(3c)
data returned by stat system	call. stat:	stat(5)
malloc, free, realloc,	calloc: main memory allocator.	malloc(3c)
main/ malloc, free, realloc,	calloc, mallopt, mallinfo: fast	malloc(3x)
intro: introduction to system	calls and error numbers.	intro(2)
	cat: concatenate and print files.	cat(1)
	cb: C program beautifier.	cb(1)
	cd: change working directory.	cd(1)
allocation. brk, sbrk:	change data segment space	brk(2)
chmod:	change mode of file.	chmod(2)
nice:	change priority of a process.	nice(2)
cd:	change working directory.	cd(1)
chdir:	change working directory.	chdir(2)
pipe: create an interprocess	channel.	pipe(2)
ungetc: push	character back into input stream.	ungetc(3s)
getc, getchar, fgetc, getw: get	character or word from a stream.	getc(3s)
putc, putchar, fputc, putw: put	character or word on a stream.	putc(3s)
ascii: map of ASCII	character set.	ascii(5)
iscntrl, isascii: classify	characters. /isprint, isgraph,	ctype(3c)

Permuted Index

tr: translate characters. . . . . tr(1)
chdir: change working directory. . . . . chdir(2)
lint: a C program checker. . . . . lint(1)
actions. setpause, checkpause: control signal . . . . . setpause(2)
terminate. wait: wait for child process to stop or . . . . . wait(2)
chmod: change mode of file. . . . . chmod(2)
isgraph, iscntrl, isascii: classify characters. /isprint, ... ctype(3c)
inquiries. ferror, feof, clearerr, fileno: stream status . . . . . ferror(3s)
close: close a file descriptor. . . . . close(2)
close: close a file descriptor. . . . . close(2)
fclose, fflush: close or flush a stream. . . . . fclose(3s)
cmp: compare two files. . . . . cmp(1)
test: condition evaluation command. . . . . test(1)
/shell, the standard/restricted command programming language. . . . . sh(1)
programs. intro: introduction to commands and application . . . . . intro(1)
diff: differential file comparator. . . . . diff(1)
cmp: compare two files. . . . . cmp(1)
yacc: yet another compiler-compiler. . . . . yacc(1)
wait: await completion of process. . . . . wait(1)
cat: concatenate and print files. . . . . cat(1)
test: condition evaluation command. . . . . test(1)
ls: list contents of directory. . . . . ls(1)
csplit: context split. . . . . csplit(1)
fcntl: file control. . . . . fcntl(2)
fcntl: file control options. . . . . fcntl(5)
setpause, checkpause: control signal actions. . . . . setpause(2)
scanf, fscanf, sscanf: convert formatted input. . . . . scanf(3s)
strtol, atol, atoi: convert string to integer. . . . . strtol(3c)
cp, mv: copy or move files. . . . . cp(1)
wc: word count. . . . . wc(1)
cp, mv: copy or move files. . . . . cp(1)
cpp: the C language preprocessor. . . . . cpp(1)
rewrite an existing one. creat: create a new file or . . . . . creat(2)
existing one. creat: create a new file or rewrite an . . . . . creat(2)
fork: create a new process. . . . . fork(2)
pipe: create an interprocess channel. . . . . pipe(2)
umask: set file mode creation mask. . . . . umask(2)
csplit: context split. . . . . csplit(1)
uname: get name of current UNIX system. . . . . uname(2)
each line of a file. cut: cut out selected fields of . . . . . cut(1)
line of a file. cut: cut out selected fields of each . . . . . cut(1)
call. stat: data returned by stat system . . . . . stat(5)
brk, sbrk: change data segment space allocation. . . . . brk(2)
types: primitive system data types. . . . . types(5)
basename, dirname: deliver portions of pathnames. . . . . basename(1)
tail: deliver the last part of a file. . . . . tail(1)
close: close a file descriptor. . . . . close(2)
dup: duplicate an open file descriptor. . . . . dup(2)
file. access: determine accessibility of a . . . . . access(2)
comparator. diff: differential file . . . . . diff(1)
diff: differential file comparator. . . . . diff(1)
dir: format of directories. . . . . dir(4)
dir: format of directories. . . . . dir(4)
rm, rmdir: remove files or directories. . . . . rm(1)
cd: change working directory. . . . . cd(1)
chdir: change working directory. . . . . chdir(2)
ls: list contents of directory. . . . . ls(1)
mkdir: make a directory. . . . . mkdir(1)
unlink: remove directory entry. . . . . unlink(2)
pwd: working directory name. . . . . pwd(1)
mknod: make a directory or a special file. . . . . mknod(2)
pathnames. basename, dirname: deliver portions of . . . . . basename(1)

od: octal	dump.	od(1)
descriptor.	dup: duplicate an open file	dup(2)
descriptor. dup:	duplicate an open file	dup(2)
echo:	echo arguments.	echo(1)
	echo: echo arguments.	echo(1)
	ed, red: text editor.	ed(1)
ed, red: text	editor.	ed(1)
ld: link	editor for PERQ C object files.	ld(1)
pattern. grep,	egrep, fgrep: search a file for a	grep(1)
unlink: remove directory	entry.	unlink(2)
	environ: user environment.	environ(5)
environ: user	environment.	environ(5)
getenv: return value for	environment name.	getenv(3c)
system error messages. perror,	errno, sys_errlist, sys_nerr:	perror(3c)
sys_errlist, sys_nerr: system	error messages. perror, errno,	perror(3c)
introduction to system calls and	error numbers. intro:	intro(2)
expression. expr:	evaluate arguments as an	expr(1)
test: condition	evaluation command.	test(1)
execlp, execvp: execute a file.	execl, execv, execle, execve,	exec(2)
execute a file. execl, execv,	execle, execve, execlp, execvp:	exec(2)
execl, execv, execle, execve,	execlp, execvp: execute a file.	exec(2)
execle, execve, execlp, execvp:	execute a file. execl, execv,	exec(2)
execvp: execute a file. execl,	execv, execle, execve, execlp,	exec(2)
file. execl, execv, execle,	execve, execlp, execvp: execute a	exec(2)
execv, execle, execve, execlp,	execvp: execute a file. execl,	exec(2)
create a new file or rewrite an	existing one. creat:	creat(2)
	exit, _exit: terminate process.	exit(2)
exit,	_exit: terminate process.	exit(2)
expression.	expr: evaluate arguments as an	expr(1)
expr: evaluate arguments as an	expression.	expr(1)
true,	false: provide truth values.	true(1)
/calloc, mallopt, mallinfo:	fast main memory allocator.	malloc(3x)
stream.	fclose, fflush: close or flush a	fclose(3s)
	fcntl: file control.	fcntl(2)
	fcntl: file control options.	fcntl(5)
fopen, freopen,	fdopen: open a stream.	fopen(3s)
status inquiries. ferror,	feof, clearerr, fileno: stream	ferror(3s)
stream status inquiries.	ferror, feof, clearerr, fileno:	ferror(3s)
fclose,	fflush: close or flush a stream.	fclose(3s)
word from a/ getc, getchar,	fgetc, getw: get character or	getc(3s)
stream. gets,	fgets: get a string from a	gets(3s)
pattern. grep, egrep,	fgrep: search a file for a	grep(1)
cut: cut out selected	fields of each line of a file.	cut(1)
determine accessibility of a	file. access:	access(2)
chmod: change mode of	file.	chmod(2)
selected fields of each line of a	file. cut: cut out	cut(1)
execve, execlp, execvp: execute a	file. execl, execv, execle,	exec(2)
link: link to a	file.	link(2)
make a directory or a special	file. mknod:	mknod(2)
files or subsequent lines of one	file. /same lines of several	paste(1)
read: read from	file.	read(2)
tail: deliver the last part of a	file.	tail(1)
write: write on a	file.	write(2)
diff: differential	file comparator.	diff(1)
fcntl:	file control.	fcntl(2)
fcntl:	file control options.	fcntl(5)
close: close a	file descriptor.	close(2)
dup: duplicate an open	file descriptor.	dup(2)
grep, egrep, fgrep: search a	file for a pattern.	grep(1)
intro: introduction to	file formats.	intro(4)
umask: set	file mode creation mask.	umask(2)
creat: create a new	file or rewrite an existing one.	creat(2)

Permuted Index

lseek: move read/write	file pointer.	lseek(2)
rewind, ftell: reposition a	file pointer in a stream. fseek,	fseek(3s)
stat, fstat: get	file status.	stat(2)
mktemp: make a unique	filename.	mktemp(3c)
ferror, feof, clearerr,	fileno: stream status inquiries.	ferror(3s)
cat: concatenate and print	files.	cat(1)
cmp: compare two	files.	cmp(1)
cp, mv: copy or move	files.	cp(1)
ld: link editor for PERQ C object	files.	ld(1)
pr: print	files.	pr(1)
sort: sort and/or merge	files.	sort(1)
rm, rmdir: remove	files or directories.	rm(1)
/merge same lines of several	files or subsequent lines of one/	paste(1)
nl: line numbering	filter.	nl(1)
tee: pipe	fitting.	tee(1)
ldexp, modf: manipulate parts of	floating-point numbers. frexp,	frexp(3c)
fclose, fflush: close or	flush a stream.	fclose(3s)
stream.	fopen, freopen, fdopen: open a	fopen(3s)
	fork: create a new process.	fork(2)
inode:	format of an i-node.	inode(4)
dir:	format of directories.	dir(4)
intro: introduction to file	formats.	intro(4)
scanf, fscanf, sscanf: convert	formatted input.	scanf(3s)
printf, fprintf, sprintf: print	formatted output.	printf(3s)
/vfprintf, vsprintf: print	formatted output of a varargs/	vprintf(3s)
/vfprintf, vsprintf: print	formatted output of a varargs/	vprintf(3x)
output. printf,	fprintf, sprintf: print formatted	printf(3s)
word on a stream. putc, putchar,	fputc, putw: put character or	putc(3s)
puts,	fputs: put a string on a stream.	puts(3s)
input/output.	fread, fwrite: binary	fread(3s)
memory allocator. malloc,	free, realloc, calloc: main	malloc(3c)
mallinfo: fast main/ malloc,	free. realloc, calloc, mallopt,	malloc(3x)
fopen,	freopen, fdopen: open a stream.	fopen(3s)
parts of floating-point numbers.	frexp, ldexp, modf: manipulate	frexp(3c)
input. scanf,	fscanf, sscanf: convert formatted	scanf(3s)
a file pointer in a stream.	fseek, rewind, ftell: reposition	fseek(3s)
stat,	fstat: get file status.	stat(2)
in a stream. fseek, rewind,	ftell: reposition a file pointer	fseek(3s)
fread,	fwrite: binary input/output.	fread(3s)
lexical tasks. lex:	generate programs for simple	lex(1)
rand, srand: simple random-number	generator.	rand(3c)
character or word from a stream.	getc, getchar, fgetc, getw: get	getc(3s)
character or word from a/ getc,	getchar, fgetc, getw: get	getc(3s)
environment name.	getenv: return value for	getenv(3c)
process group, and/ getpid,	getpgrp, getppid: get process,	getpid(2)
process, process group, and/	getpid, getpgrp, getppid: get	getpid(2)
group, and/ getpid, getpgrp,	getppid: get process, process	getpid(2)
stream.	gets, fgets: get a string from a	gets(3s)
a stream. getc, getchar, fgetc,	getw: get character or word from	getc(3s)
setjmp, longjmp: non-local	goto.	setjmp(3c)
for a pattern.	grep, egrep, fgrep: search a file	grep(1)
/getppid: get process, process	group, and parent process IDs.	getpid(2)
send a signal to a process or a	group of processes. kill:	kill(2)
maintain, update, and regenerate	groups of programs. make:	make(1)
varargs:	handle variable argument lists.	varargs(5)
inode: format of an	i-node.	inode(4)
	inode: format of an i-node.	inode(4)
fscanf, sscanf: convert formatted	input. scanf,	scanf(3s)
ungetc: push character back into	input stream.	ungetc(3s)
fread, fwrite: binary	input/output.	fread(3s)
stdio: standard buffered	input/output package.	stdio(3s)
clearerr, fileno: stream status	inquiries. ferror, feof,	ferror(3s)

atol, atoi: convert string to	integer. strtol.	strtol(3c)
pipe: create an	interprocess channel.	pipe(2)
and application programs.	intro: introduction to commands	intro(1)
formats.	intro: introduction to file	intro(4)
miscellany.	intro: introduction to	intro(5)
specialized routines and/	intro: introduction to	intro(3x)
subroutines and libraries.	intro: introduction to	intro(3)
calls and error numbers.	intro: introduction to system	intro(2)
application programs. intro:	introduction to commands and	intro(1)
intro:	introduction to file formats.	intro(4)
intro:	introduction to miscellany.	intro(5)
routines and libraries. intro:	introduction to specialized	intro(3x)
libraries. intro:	introduction to subroutines and	intro(3)
error numbers. intro:	introduction to system calls and	intro(2)
/islower, isdigit, isxdigit,	isalnum, isspace, ispunct,/	ctype(3c)
isdigit, isxdigit, isalnum,/	isalpha, isupper, islower, ...	ctype(3c)
/isprint, isgraph, iscntrl,	isascii: classify characters.	ctype(3c)
/ispunct, isprint, isgraph,	iscntrl, isascii: classify/	ctype(3c)
isalpha, isupper, islower,	isdigit, isxdigit, isalnum,/	ctype(3c)
/isspace, ispunct, isprint,	isgraph, iscntrl, isascii:/	ctype(3c)
isalnum,/ isalpha, isupper,	islower, isdigit, isxdigit, ...	ctype(3c)
/isalnum, isspace, ispunct,	isprint, isgraph, iscntrl,/	ctype(3c)
/isxdigit, isalnum, isspace,	ispunct, isprint, isgraph,/	ctype(3c)
/isdigit, isxdigit, isalnum,	isspace, ispunct, isprint,/	ctype(3c)
isxdigit, isalnum,/ isalpha,	isupper, islower, isdigit, ...	ctype(3c)
/isupper, islower, isdigit,	isxdigit, isalnum, isspace,/	ctype(3c)
or a group of processes.	kill: send a signal to a process	kill(2)
	kill: terminate a process.	kill(1)
pattern scanning and processing	language. awk:	awk(1)
command programming	language. /standard/restricted	sh(1)
cpp: the C	language preprocessor.	cpp(1)
files.	ld: link editor for PERQ C object	ld(1)
floating-point numbers. frexp,	ldexp, modf: manipulate parts of	frexp(3c)
lexical tasks.	lex: generate programs for simple	lex(1)
lex: generate programs for simple	lexical tasks.	lex(1)
introduction to subroutines and	libraries. intro:	intro(3)
to specialized routines and	libraries. intro: introduction	intro(3x)
nl:	line numbering filter.	nl(1).
cut out selected fields of each	line of a file. cut:	cut(1)
of several files or subsequent	lines of one file. /same lines	paste(1)
subsequent/ paste: merge same	lines of several files or	paste(1)
files. ld:	link editor for PERQ C object	ld(1)
	link: link to a file.	link(2)
link:	link to a file.	link(2)
	lint: a C program checker.	lint(1)
output of a varargs argument	list. /vsprintf: print formatted	vprintf(3s)
output of a varargs argument	list. /vsprintf: print formatted	vprintf(3x)
ls:	list contents of directory.	ls(1)
varargs: handle variable argument	lists.	varargs(5)
setjmp,	longjmp: non-local goto.	setjmp(3c)
	ls: list contents of directory.	ls(1)
pointer.	lseek: move read/write file	lseek(2)
values:	machine-dependent values.	values(5)
malloc, free, realloc, calloc:	main memory allocator.	malloc(3c)
calloc, mallopt, mallinfo: fast	main memory allocator. /realloc, ...	malloc(3x)
groups of programs. make:	maintain, update, and regenerate	make(1)
regenerate groups of programs.	make: maintain, update, and	make(1)
/free, realloc, calloc, mallopt,	mallinfo: fast main memory/	malloc(3x)
main memory allocator.	malloc, free, realloc, calloc:	malloc(3c)
mallopt, mallinfo: fast main/	malloc, free, realloc, calloc, ...	malloc(3x)
malloc, free, realloc, calloc,	mallopt, mallinfo: fast main/	malloc(3x)
frexp, ldexp, modf:	manipulate parts of/	frexp(3c)

Permuted Index

ascii:	map of ASCII character set.	ascii(5)
umask: set file mode creation	mask.	umask(2)
memset: memory operations.	memccpy, memchr, memcmp, memcpy,	memory(3c)
memory operations. memccpy,	memchr, memcmp, memcpy, memset:	memory(3c)
operations. memccpy, memchr,	memcmp, memcpy, memset: memory	memory(3c)
memccpy, memchr, memcmp,	memcpy, memset: memory/	memory(3c)
free, realloc, calloc: main	memory allocator. malloc,	malloc(3c)
mallopt, mallinfo: fast main	memory allocator. /calloc,	malloc(3x)
memchr, memcmp, memcpy, memset:	memory operations. memccpy,	memory(3c)
memccpy, memchr, memcmp, memcpy,	memset: memory operations.	memory(3c)
sort: sort and/or	merge files.	sort(1)
or subsequent lines of/ paste:	merge same lines of several files	paste(1)
sys_nerr: system error	messages. /errno, sys_errlist,	perror(3c)
intro: introduction to	miscellany.	intro(5)
	mkdir: make a directory.	mkdir(1)
special file.	mknod: make a directory or a	mknod(2)
	mktemp: make a unique filename.	mktemp(3c)
umask: set file	mode creation mask.	umask(2)
chmod: change	mode of file.	chmod(2)
floating-point/ frexp, ldexp,	modf: manipulate parts of	frexp(3c)
cp, mv: copy or	move files.	cp(1)
lseek:	move read/write file pointer.	lseek(2)
cp,	mv: copy or move files.	cp(1)
return value for environment	name. getenv:	getenv(3c)
pwd: working directory	name.	pwd(1)
uname: get	name of current UNIX system.	uname(2)
process.	nice: change priority of a	nice(2)
	nl: line numbering filter.	nl(1)
setjmp, longjmp:	non-local goto.	setjmp(3c)
nl: line	numbering filter.	nl(1)
parts of floating-point	numbers. /ldexp, modf: manipulate	frexp(3c)
to system calls and error	numbers. intro: introduction	intro(2)
ld: link editor for PERQ C	object files.	ld(1)
od:	octal dump.	od(1)
	od: octal dump.	od(1)
a new file or rewrite an existing	one. creat: create	creat(2)
fopen, freopen, fdopen:	open a stream.	fopen(3s)
dup: duplicate an	open file descriptor.	dup(2)
open:	open for reading or writing.	open(2)
writing.	open: open for reading or	open(2)
memcmp, memcpy, memset: memory	operations. memccpy, memchr,	memory(3c)
strspn, strcspn, strtok: string	operations. /strrchr, strpbrk,	string(3c)
fcntl: file control	options.	fcntl(5)
fprintf, sprintf: print formatted	output. printf,	printf(3s)
list. /vsprintf: print formatted	output of a varargs argument	vprintf(3s)
list. /vsprintf: print formatted	output of a varargs argument	vprintf(3x)
standard buffered input/output	package. stdio:	stdio(3s)
get process, process group, and	parent process IDs. /getppid:	getpid(2)
tail: deliver the last	part of a file.	tail(1)
frexp, ldexp, modf: manipulate	parts of floating-point numbers.	frexp(3c)
several files or subsequent/	paste: merge same lines of	paste(1)
dirname: deliver portions of	pathnames. basename,	basename(1)
egrep, fgrep: search a file for a	pattern. grep,	grep(1)
language. awk:	pattern scanning and processing	awk(1)
signal.	pause: suspend process until	pause(2)
sys_nerr: system error messages.	perror, errno, sys_errlist,	perror(3c)
channel.	pipe: create an interprocess	pipe(2)
tee:	pipe fitting.	tee(1)
lseek: move read/write file	pointer.	lseek(2)
rewind, ftell: reposition a file	pointer in a stream. fseek,	fseek(3s)
basename, dirname: deliver	portions of pathnames.	basename(1)
	pr: print files.	pr(1)

cpp: the C language	preprocessor.	cpp(1)
types:	primitive system data types.	types(5)
cat: concatenate and	print files.	cat(1)
pr:	print files.	pr(1)
printf, fprintf, sprintf:	print formatted output.	printf(3s)
vprintf, vfprintf, vsprintf:	print formatted output of a/	vprintf(3s)
vprintf, vfprintf, vsprintf:	print formatted output of a/	vprintf(3x)
formatted output.	printf, fprintf, sprintf: print	printf(3s)
nice: change	priority of a process.	nice(2)
exit, _exit: terminate	process.	exit(2)
fork: create a new	process.	fork(2)
kill: terminate a	process.	kill(1)
nice: change priority of a	process.	nice(2)
wait: await completion of	process.	wait(1)
process group, and parent	process IDs. /get process,	getpid(2)
/getpgrp, getppid: get process,	process group, and parent/	getpid(2)
kill: send a signal to a	process or a group of processes.	kill(2)
getpid, getpgrp, getppid: get	process, process group, and/	getpid(2)
wait: wait for child	process to stop or terminate.	wait(2)
pause: suspend	process until signal.	pause(2)
signal to a process or a group of	processes. kill: send a	kill(2)
awk: pattern scanning and	processing language.	awk(1)
cb: C	program beautifier.	cb(1)
lint: a C	program checker.	lint(1)
the standard/restricted command	programming language. /shell,	sh(1)
to commands and application	programs. intro: introduction	intro(1)
update, and regenerate groups of	programs. make: maintain,	make(1)
tasks. lex: generate	programs for simple lexical	lex(1)
true, false:	provide truth values.	true(1)
stream. ungetc:	push character back into input	ungetc(3s)
puts, fputs:	put a string on a stream.	puts(3s)
putc, putchar, fputc, putw:	put character or word on a/	putc(3s)
character or word on a stream.	putc, putchar, fputc, putw: put	putc(3s)
character or word on a/ putc,	putchar, fputc, putw: put	putc(3s)
stream.	puts, fputs: put a string on a	puts(3s)
stream. putc, putchar, fputc,	putw: put character or word on a	putc(3s)
	pwd: working directory name.	pwd(1)
generator.	rand, srand: simple random-number	rand(3c)
rand, srand: simple	random-number generator.	rand(3c)
read:	read from file.	read(2)
	read: read from file.	read(2)
open: open for	reading or writing.	open(2)
lseek: move	read/write file pointer.	lseek(2)
allocator. malloc, free,	realloc, calloc: main memory	malloc(3c)
mallinfo: fast/ malloc, free,	realloc, calloc, mallopt,	malloc(3x)
signal: specify what to do upon	receipt of a signal.	signal(2)
ed,	red: text editor.	ed(1)
make: maintain, update, and	regenerate groups of programs.	make(1)
unlink:	remove directory entry.	unlink(2)
rm, rmdir:	remove files or directories.	rm(1)
stream. fseek, rewind, ftell:	reposition a file pointer in a	fseek(3s)
name. getenv:	return value for environment	getenv(3c)
stat: data	returned by stat system call.	stat(5)
pointer in a stream. fseek,	rewind, ftell: reposition a file	fseek(3s)
creat: create a new file or	rewrite an existing one.	creat(2)
directories.	rm, rmdir: remove files or	rm(1)
directories. rm,	rmdir: remove files or	rm(1)
introduction to specialized	routines and libraries. intro:	intro(3x)
standard/restricted command/ sh,	rsh: shell, the	sh(1)
allocation. brk,	sbrk: change data segment space	brk(2)
formatted input.	scanf, fscanf, sscanf: convert	scanf(3s)
awk: pattern	scanning and processing language.	awk(1)

Permuted Index

grep, egrep, fgrep: search a file for a pattern. . . . . grep(1)
brk, sbrk: change data segment space allocation. . . . . brk(2)
file. cut: cut out selected fields of each line of a . . . . . cut(1)
group of processes. kill: send a signal to a process or a . . . . . kill(2)
ascii: map of ASCII character set. . . . . ascii(5)
umask: set file mode creation mask. . . . . umask(2)
to a stream. setbuf, setvbuf: assign buffering . . . . . setbuf(3s)
setjmp, longjmp: non-local goto. . . . . setjmp(3c)
signal actions. setpause, checkpause: control . . . . . setpause(2)
stream. setbuf, setvbuf: assign buffering to a . . . . . setbuf(3s)
of/ paste: merge same lines of several files or subsequent lines . . . . . paste(1)
standard/restricted command/ sh, rsh: shell, the . . . . . sh(1)
command programming/ sh, rsh: shell, the standard/restricted . . . . . sh(1)
pause: suspend process until signal. . . . . pause(2)
what to do upon receipt of a signal. signal: specify . . . . . signal(2)
setpause, checkpause: control signal actions. . . . . setpause(2)
receipt of a signal. signal: specify what to do upon . . . . . signal(2)
processes. kill: send a signal to a process or a group of . . . . . kill(2)
lex: generate programs for simple lexical tasks. . . . . lex(1)
rand, srand: simple random-number generator. . . . . rand(3c)
sort: sort and/or merge files. . . . . sort(1)
sort: sort and/or merge files. . . . . sort(1)
brk, sbrk: change data segment space allocation. . . . . brk(2)
mknod: make a directory or a special file. . . . . mknod(2)
intro: introduction to specialized routines and/ . . . . . intro(3x)
of a signal. signal: specify what to do upon receipt . . . . . signal(2)
csplit: context split. . . . . csplit(1)
printf, fprintf, sprintf: print formatted output. . . . . printf(3s)
generator. rand, srand: simple random-number . . . . . rand(3c)
scanf, fscanf, sscanf: convert formatted input. . . . . scanf(3s)
package. stdio: standard buffered input/output . . . . . stdio(3s)
programming/ sh, rsh: shell, the standard/restricted command . . . . . sh(1)
system call. stat: data returned by stat . . . . . stat(5)
stat, fstat: get file status. . . . . stat(2)
stat: data returned by stat system call. . . . . stat(5)
stat, fstat: get file status. . . . . stat(2)
feof, clearerr, fileno: stream status inquiries. ferror, ... ferror(3s)
input/output package. stdio: standard buffered . . . . . stdio(3s)
wait: wait for child process to stop or terminate. . . . . wait(2)
strcpy, strncpy, strlen, strchr,/ strcat, strncat, strcmp, strncmp, ... string(3c)
/strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn,/ . . . . . string(3c)
strlen, strchr,/ strcat, strncat, strcmp, strncmp, strcpy, strncpy, ... string(3c)
strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr,/ . . . . . string(3c)
strchr, strrchr, strpbrk, strspn, strcspn, strtok: string/ /strlen, ... string(3c)
fclose, fflush: close or flush a stream. . . . . fclose(3s)
fopen, freopen, fdopen: open a stream. . . . . fopen(3s)
reposition a file pointer in a stream. fseek, rewind, ftell: . . . . . fseek(3s)
get character or word from a stream. /getchar, fgetc, getw: . . . . . getc(3s)
gets, fgets: get a string from a stream. . . . . gets(3s)
putw: put character or word on a stream. putc, putchar, fputc, ... putc(3s)
puts, fputs: put a string on a stream. . . . . puts(3s)
setvbuf: assign buffering to a stream. setbuf, ... setbuf(3s)
push character back into input stream. ungetc: . . . . . ungetc(3s)
ferror, feof, clearerr, fileno: stream status inquiries. . . . . ferror(3s)
gets, fgets: get a string from a stream. . . . . gets(3s)
puts, fputs: put a string on a stream. . . . . puts(3s)
strpbrk, strspn, strcspn, strtok: string operations. /strrchr, ... string(3c)
strtol, atol, atoi: convert string to integer. . . . . strtol(3c)
/strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk,/ . . . . . string(3c)
strncpy, strlen, strchr,/ strcat, strncat, strcmp, strncmp, strcpy, ... string(3c)
strchr,/ strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, ... string(3c)
/strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr,/ . . . . . string(3c)

/strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok:/ . . . . . string(3c)
/strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn,/ . . . . . string(3c)
/strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok: string/ . . . . . string(3c)
/strpbrk, strspn, strcspn, strtok: string operations. . . . . string(3c)
string to integer. strtol, atol, atoi: convert . . . . . strtol(3c)
intro: introduction to subroutines and libraries. . . . . intro(3)
/same lines of several files or subsequent lines of one file. . . . . paste(1)
pause: suspend process until signal. . . . . pause(2)
swab: swap bytes. . . . . swab(3c)
swab: swap bytes. . . . . swab(3c)
error messages. perror, errno, sys_errlist, sys_nerr: system . . . . . perror(3c)
perror, errno, sys_errlist, sys_nerr: system error messages. . . . . perror(3c)
uname: get name of current UNIX system. . . . . uname(2)
stat: data returned by stat system call. . . . . stat(5)
intro: introduction to system calls and error numbers. . . . . intro(2)
types: primitive system data types. . . . . types(5)
errno, sys_errlist, sys_nerr: system error messages. perror, ... perror(3c)
file. tail: deliver the last part of a . . . . . tail(1)
programs for simple lexical tasks. lex: generate . . . . . lex(1)
tee: pipe fitting. . . . . tee(1)
wait for child process to stop or terminate. wait: . . . . . wait(2)
kill: terminate a process. . . . . kill(1)
exit, _exit: terminate process. . . . . exit(2)
command. test: condition evaluation . . . . . test(1)
ed, red: text editor. . . . . ed(1)
time: get time. . . . . time(2)
time: get time. . . . . time(2)
tr: translate characters. . . . . tr(1)
tr: translate characters. . . . . tr(1)
values. true, false: provide truth . . . . . true(1)
true, false: provide truth values. . . . . true(1)
types: primitive system data types. . . . . types(5)
types. types: primitive system data . . . . . types(5)
mask. umask: set file mode creation . . . . . umask(2)
system. uname: get name of current UNIX . . . . . uname(2)
input stream. ungetc: push character back into . . . . . ungetc(3s)
mktemp: make a unique filename. . . . . mktemp(3c)
unlink: remove directory entry. . . . . unlink(2)
pause: suspend process until signal. . . . . pause(2)
programs. make: maintain, update, and regenerate groups of . . . . . make(1)
signal: specify what to do upon receipt of a signal. . . . . signal(2)
environ: user environment. . . . . environ(5)
getenv: return value for environment name. . . . . getenv(3c)
true, false: provide truth values. . . . . true(1)
values: machine-dependent values. . . . . values(5)
values: machine-dependent values. . . . . values(5)
print formatted output of a varargs argument list. /vsprintf: . . . . . vprintf(3s)
print formatted output of a varargs argument list. /vsprintf: . . . . . vprintf(3x)
lists. varargs: handle variable argument . . . . . varargs(5)
varargs: handle variable argument lists. . . . . varargs(5)
formatted output of a/ vprintf, vfprintf, vsprintf: print . . . . . vprintf(3s)
formatted output of a/ vprintf, vfprintf, vsprintf: print . . . . . vprintf(3x)
print formatted output of a/ vprintf, vfprintf, vsprintf: . . . . . vprintf(3s)
print formatted output of a/ vprintf, vfprintf, vsprintf: . . . . . vprintf(3x)
of a varargs/ vprintf, vfprintf, vsprintf: print formatted output . . . . . vprintf(3s)
of a varargs/ vprintf, vfprintf, vsprintf: print formatted output . . . . . vprintf(3x)
process. wait: await completion of . . . . . wait(1)
terminate. wait: wait for child process to stop or . . . . . wait(2)
stop or terminate. wait: wait for child process to . . . . . wait(2)
wc: word count. . . . . wc(1)
wc: word count. . . . . wc(1)
fgetc, getw: get character or word from a stream. /getchar, ... getc(3s)

Permuted Index

fputc, putw: put character or word on a stream. putc, putchar, ... putc(3s)
cd: change working directory. . . . . cd(1)
chdir: change working directory. . . . . chdir(2)
pwd: working directory name. . . . . pwd(1)
write: write on a file. . . . . write(2)
write: write on a file. . . . . write(2)
open: open for reading or writing. . . . . open(2)
compiler-compiler. yacc: yet another . . . . . yacc(1)
yacc: yet another compiler-compiler. . . . . yacc(1)